

Lecture Notes in Computer Science

1954

Warren A. Hunt, Jr. Steven D. Johnson (Eds.)

Formal Methods in Computer-Aided Design

Third International Conference, FMCAD 2000
Austin, TX, USA, November 2000
Proceedings



Springer

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

1954

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Warren A. Hunt, Jr. Steven D. Johnson (Eds.)

Formal Methods in Computer-Aided Design

Third International Conference, FMCAD 2000
Austin, TX, USA, November 1-3, 2000
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Warren A. Hunt, Jr.
IBM Corporation, Austin Research Laboratories
Mail Stop 9460, Building 904
11501 Burnet Road, Austin, Texas 78758, USA
E-mail: whunt@austin.ibm.com

Steven D. Johnson
Indiana University, Computer Science Department
Lindley Hall, 150 W. Woodlawn Avenue
Bloomington, Indiana 47405-7104, USA
E-mail: sjohnson@cs.indiana.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Formal methods in computer aided design : third international
conference ; proceedings / FMCAD 2000, Austin, Texas, USA, November
1 - 3, 2000. Warren A. Hunt, jr. ; Steven D. Johnson (ed.). - Berlin ;
Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ;
Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1954)
ISBN 3-540-41219-0

CR Subject Classification (1998): B.1.2, B.1.4, B.2.2-3, B.6.2-3, B.7.2-3, F.3.1,
F.4.1, I.2.3, D.2.4, J.6

ISSN 0302-9743

ISBN 3-540-41219-0 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH
© Springer-Verlag Berlin Heidelberg 2000
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN: 10780987 06/3142 5 4 3 2 1 0

Preface

The biannual *Formal Methods in Computer Aided Design* conference (FMCAD 2000) is the third in a series of conferences under that title devoted to the use of discrete mathematical methods for the analysis of computer hardware and software. The work reported in this book describes the use of modeling languages and their associated automated analysis tools to specify and verify computing systems.

Functional verification has become one of the principal costs in a modern computer design effort. In addition, verification of circuit models, timing, power, etc., requires even more effort. FMCAD provides a venue for academic and industrial researchers and practitioners to share their ideas and experiences of using discrete mathematical modeling and verification. It is noted with interest by the conference chairmen how this area has grown from just a few people 15 years ago to a vibrant area of research, development, and deployment. It is clear that these methods are helping reduce the cost of designing computing systems. As an example of this potential cost reduction, we have invited David Russinoff of Advanced Micro Devices, Inc. to describe his verification of floating-point algorithms being used in AMD microprocessors. The program includes 30 regular presentations selected from 63 submitted papers.

The FMCAD conference has a long history dating back to 1984, when the earliest meetings on this topic occurred. A series of workshops sponsored by IFIP WG10.2 were held in Darmstadt (1984, org. Hans Eveking), Edinburgh (1985, org. George J. Milne and P.A. Subrahmanyam), Grenoble (1986, org. Dominique Borriane), Glasgow (1988, org. George J. Milne), Leuven (1989, org. Luc Claessen), and Miami (1990, org. P.A. Subrahmanyam). FMCAD originally had the name *Theorem Provers in Circuit Design*. TPCD meetings were held in Nijmegen (1992, org. Raymond T. Boute, Thomas Melham, and Victoria Stavridou) and Bad Herrenalb (1994, org. Thomas Kropf and Ramayya Kumar). Renamed *Formal Methods in Computer Aided Design*, the venue was changed to San Jose for the next two meetings (1996, org. Albert Camilleri and Mandayam Srivas, and 1998, org. Ganesh Goplakrishnan and Phillip J. Windley). FMCAD 2000 was held in Austin. FMCAD alternates with the biannual conference on *Correct Hardware Design and Verification Methods*. CHARME originated with a research presentation of the ESPRIT group “CHARME” at Torino. Subsequent conferences were held at Arles (1993, org. George J. Milne and Laurence Pierre), Frankfurt (1995, org. Hans Eveking and Paolo Camurati), Montreal (1997, org. Hon F. Li and David K. Probst), and Bad Herrenalb (1999, org. Thomas Kropf and Laurence Pierre).

The organizers are grateful to Advanced Micro Devices, Inc., Cadence Design Systems, Inc., Compaq Computer Corp., IBM Corp., Intel Corp., Prover Technology AB, Real Intent Corp., Synopsys, Inc., and Xilinx Inc., for their financial sponsorship, which considerably eased the organization of the conference.

Dan Elgin and Jo Moore are to be thanked for their tireless effort; they kept us on an organized and orderly path.

November 2000

Warren A. Hunt, Jr.
Steven D. Johnson

Organization

FMCAD 2000 was organized and held in Austin, Texas, U.S.A., at Austin's Marriott at the Capitol. An ACL2 workshop and a tutorial and workshop on Formal Specification and Verification Methods for Shared Memory Systems were also held in conjunction with FMCAD 2000.

Program Committee

Conference Chairs: Warren A. Hunt, Jr. (IBM Austin Research Lab, USA)

Steven D. Johnson (Indiana University, USA)

Mark Aagaard (Intel Corporation, USA)

Dominique Borri  ne (Laboratoire TIMA and Universit   Joseph Fourier, France)

Randy Bryant (Carnegie Mellon University, USA)

Albert Camilleri (Hewlett-Packard Co., USA)

Eduard Cerny (Universit   de Montr  al, Canada)

Shiu-Kai Chin (Syracuse University, USA)

Luc Claesen (LCI SMARTpen N.V. & K.U.Leuven, Belgium)

Edmund Clarke (Carnegie Mellon University, USA)

David L. Dill (Stanford University, USA)

Hans Eveking (Darmstadt University of Technology, Germany)

Limor Fix (Intel Corporation, Israel)

Masahiro Fujita (University of Tokyo, Japan)

Steven German (IBM T.J. Watson Research Center, USA)

Ganesh Gopalakrishnan (University of Utah, USA)

Mike Gordon (Cambridge University, UK)

Yuri Gurevich (Microsoft Research, USA)

Kieth Hanna (University of Kent, Great Britain)

Alan Hu (University of British Columbia, Canada)

Damir Jamsek (IBM Austin Research Lab, USA)

Matt Kaufmann (Advanced Micro Devices, Inc., USA)

Thomas Kropf (Robert Bosch GmbH and University of T  bingen, Germany)

Andreas Kuehlmann (Cadence Design Systems, Inc., USA)

John Launchbury (Oregon Graduate Institute, USA)

Tim Leonard (Compaq Computer Corp., USA)

Ken McMillan (Cadence Berkeley Labs, USA)

Tom Melham (University of Glasgow, Scotland)

Paul Miner (NASA Langley Research Center, USA)

John O'Leary (Intel Corp., USA)

Laurence Pierre (Universit   de Provence, France)

Carl Pixley (Motorola, Inc., USA)

Amir Pnueli (Technion, Israel)

Rajeev Ranjan (Real Intent Corp., USA)

VIII Organization

David Russinoff (Advanced Micro Devices, Inc., USA)

Mary Sheeran (Chalmers Univ. of Technology and Prover Technology, Sweden)

Anna Slobodova (Compaq Computer Corp., USA)

Mandayam Srivas (SRI International, USA)

Victoria Stavridou (SRI International, USA)

Ranga Vemuri (University of Cincinnati, USA)

Matthew Wilding (Rockwell Collins, Inc., USA)

Phillip J. Windley (Brigham Young University, USA)

Additional Referees

Tamarah Arons

Annette Bunker

Venkatesh Choppella

Bruno Dutertre

Dana Fisman

Rob Gerth

David Greve

Dirk Hoffmann

Ravi Hosabettu

Michael D. Jones

Nazanin Mansouri

Nir Piterman

Rajesh Radhakrishnan

Sriram Rajamani

Vanderlei Rodrigues

Hassen Saidi

Elad Shahar

Robert Shaw

Kanna Shimizu

Robert Sumners

Elena Teica

Margus Veanes

FMCAD'00 Bibliography

- [1] Warren A. Hunt, Jr. and Steven D. Johnson, editors. *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg Berlin, 2000.
- [2] David M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD AthlonTM processor. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 3–37, Heidelberg Berlin, 2000. Springer-Verlag.
- [3] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 38–55, Heidelberg Berlin, 2000. Springer-Verlag.
- [4] David Basin, Stefan Friedrich, and Sebastian Mödersheim. B2M: A semantic based tool for BLIF hardware descriptions. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 56–73, Heidelberg Berlin, 2000. Springer-Verlag.
- [5] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 74–91, Heidelberg Berlin, 2000. Springer-Verlag.
- [6] Nancy A. Day, Mark D. Aagaard, and Byron Cook. Combining stream-based and state-based verification techniques for microarchitectures. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 92–108, Heidelberg Berlin, 2000. Springer-Verlag.
- [7] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided*

Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings, volume 1954 of *Lecture Notes in Computer Science*, pages 109–126, Heidelberg Berlin, 2000. Springer-Verlag.

- [8] Panagiotis Manolios. Correctness of pipelined machines. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–143, Heidelberg Berlin, 2000. Springer-Verlag.
- [9] Rajeev Alur, Radu Grosu, and Bow-Yaw Wang. Automated refinement checking for asynchronous processes. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 144–161, Heidelberg Berlin, 2000. Springer-Verlag.
- [10] Wolfgang Reif, Jürgen Ruf, Gerhard Schellhorn, and Tobias Vollmer. Do you trust your model checker? In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 162–179, Heidelberg Berlin, 2000. Springer-Verlag.
- [11] E. Clarke, S. German, Y. Lu, H. Veith, and D. Wang. Executable protocol specification in ESL. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 180–197, Heidelberg Berlin, 2000. Springer-Verlag.
- [12] John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 218–235, Heidelberg Berlin, 2000. Springer-Verlag.
- [13] Jun Sawada and Warren A. Hunt, Jr. Hardware modeling using function encapsulation. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 236–247, Heidelberg Berlin, 2000. Springer-Verlag.
- [14] Antonio Cerone and George J. Milne. A methodology for the formal analysis of asynchronous micropipelines. In Warren A. Hunt, Jr. and Steven D.

- Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 248–264, Heidelberg Berlin, 2000. Springer-Verlag.
- [15] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O’Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 265–283, Heidelberg Berlin, 2000. Springer-Verlag.
 - [16] Nina Amla, E. Allen Emerson, Robert P. Kurshan, and Kedar S. Namjoshi. Model checking synchronous timing diagrams. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 284–299, Heidelberg Berlin, 2000. Springer-Verlag.
 - [17] Jin Hou and Eduard Cerny. Model reductions and a case study. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 300–316, Heidelberg Berlin, 2000. Springer-Verlag.
 - [18] Adilson Luiz Bonifácio and Arnaldo Vieira Moura. Modeling and parameters synthesis for an air traffic management system. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 317–334, Heidelberg Berlin, 2000. Springer-Verlag.
 - [19] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 335–353, Heidelberg Berlin, 2000. Springer-Verlag.
 - [20] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 354–371, Heidelberg Berlin, 2000. Springer-Verlag.

- [21] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 372–389, Heidelberg Berlin, 2000. Springer-Verlag.
- [22] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 390–406, Heidelberg Berlin, 2000. Springer-Verlag.
- [23] Gordon J. Pace. The semantics of Verilog using transition system combinators. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 407–424, Heidelberg Berlin, 2000. Springer-Verlag.
- [24] Gerd Ritter. Sequential equivalence checking by symbolic simulation. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 425–444, Heidelberg Berlin, 2000. Springer-Verlag.
- [25] Christoph Meinel and Christian Stangier. Speeding up image computation by using RTL information. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 445–457, Heidelberg Berlin, 2000. Springer-Verlag.
- [26] Kiyoharu Hamaguchi, Hidekazu Urushihara, and Toshinobu Kashiwabara. Symbolic checking of signal-transition consistency for verifying high-level designs. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 458–473, Heidelberg Berlin, 2000. Springer-Verlag.
- [27] Chris Wilson, David L. Dill, and Randal E. Bryant. Symbolic simulation with approximate values. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International*

Conference, *FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 474–489, Heidelberg Berlin, 2000. Springer-Verlag.

- [28] Randal E. Bryant, Pankaj Chauhan, Edmund M. Clarke, and Amit Goel. A theory of consistency for modular synchronous systems. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 490–507, Heidelberg Berlin, 2000. Springer-Verlag.
- [29] Michael Jones and Ganesh Gopalakrishnan. Verifying transaction ordering properties in unbounded bus networks through combined deductive/algorithmic methods. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 508–522, Heidelberg Berlin, 2000. Springer-Verlag.
- [30] Alex Tsow and Steven D. Johnson. Visualizing system factorizations with behavior tables. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 523–541, Heidelberg Berlin, 2000. Springer-Verlag.

Table of Contents

Invited Talk

Trends in Computing	1
<i>Mark E. Dean</i>	

Invited Paper

A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon TM Processor	3
<i>David M. Russinoff</i>	

Contributed Papers

An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps	37
<i>Roderick Bloem, Harold N. Gabow, Fabio Somenzi</i>	
Automated Refinement Checking for Asynchronous Processes	55
<i>Rajeev Alur, Radu Grosu, Bow-Yaw Wang</i>	
Border-Block Triangular Form and Conjunction Schedule in Image Computation	73
<i>In-Ho Moon, Gary D. Hachtel, Fabio Somenzi</i>	
B2M: A Semantic Based Tool for BLIF Hardware Descriptions	91
<i>David Basin, Stefan Friedrich, Sebastian Mödersheim</i>	
Checking Safety Properties Using Induction and a SAT-Solver	108
<i>Mary Sheeran, Satnam Singh, Gunnar Stålmarch</i>	
Combining Stream-Based and State-Based Verification Techniques	126
<i>Nancy A. Day, Mark D. Aagaard, Byron Cook</i>	
A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles	143
<i>Kavita Ravi, Roderick Bloem, Fabio Somenzi</i>	
Correctness of Pipelined Machines	161
<i>Panagiotis Manolios</i>	
Do You Trust Your Model Checker?	179
<i>Wolfgang Reif, Jürgen Ruf, Gerhard Schellhorn, Tobias Vollmer</i>	
Executable Protocol Specification in ESL	197
<i>Edmund M. Clarke, S. German, Y. Lu, Helmuth Veith, D. Wang</i>	

Formal Verification of Floating Point Trigonometric Functions	217
<i>John Harrison</i>	
Hardware Modeling Using Function Encapsulation	234
<i>Jun Sawada, Warren A. Hunt, Jr.</i>	
A Methodology for the Formal Analysis of Asynchronous Micropipelines . .	246
<i>Antonio Cerone, George J. Milne</i>	
A Methodology for Large-Scale Hardware Verification	263
<i>Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, Carl-Johan H. Seger</i>	
Model Checking Synchronous Timing Diagrams	283
<i>Nina Amla, E. Allen Emerson, Robert P. Kurshan, Kedar S. Namjoshi</i>	
Model Reductions and a Case Study	299
<i>Jin Hou, Eduard Cerny</i>	
Modeling and Parameters Synthesis for an Air Traffic Management System	316
<i>Adilson Luiz Bonifácio, Arnaldo Vieira Moura</i>	
Monitor-Based Formal Specification of PCI	335
<i>Kanna Shimizu, David L. Dill, Alan J. Hu</i>	
SAT-Based Image Computation with Application in Reachability Analysis	354
<i>Aarti Gupta, Zijiang Yang, Pranav Ashar, Anubhav Gupta</i>	
SAT-Based Verification without State Space Traversal	372
<i>Per Bjesse, Koen Claessen</i>	
Scalable Distributed On-the-Fly Symbolic Model Checking	390
<i>Shoham Ben-David, Tamir Heyman, Orna Grumberg, Assaf Schuster</i>	
The Semantics of Verilog Using Transition System Combinators	405
<i>Gordon J. Pace</i>	
Sequential Equivalence Checking by Symbolic Simulation	423
<i>Gerd Ritter</i>	
Speeding Up Image Computation by Using RTL Information	443
<i>Christoph Meinel, Christian Stangier</i>	
Symbolic Checking of Signal-Transition Consistency for Verifying High-Level Designs	455
<i>Kiyoharu Hamauchi, Hidekazu Urushihara, Toshinobu Kashiwabara</i>	
Symbolic Simulation with Approximate Values	470
<i>Chris Wilson, David L. Dill, Randal E. Bryant</i>	

A Theory of Consistency for Modular Synchronous Systems	486
<i>Randal E. Bryant, Pankaj Chauhan, Edmund M. Clarke, Amit Goel</i>	
Verifying Transaction Ordering Properties in Unbounded Bus Networks through Combined Deductive/Algorithmic Methods	505
<i>Michael Jones, Ganesh Gopalakrishnan</i>	
Visualizing System Factorizations with Behavior Tables	520
<i>Alex Tsow, Steven D. Johnson</i>	
Author Index	539

Applications of Hierarchical Verification in Model Checking

Robert Beers, Rajnish Ghughal, and Mark Aagaard

Performance Microprocessor Division, Intel Corporation,
RA2-401, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA.

Abstract. The LTL model checker that we use provides sound decomposition mechanisms within a purely model checking environment. We have exploited these mechanisms to successfully verify a wide spectrum of large and complex circuits. This paper describes a variety of the decomposition techniques that we have used as part of a large industrial formal verification effort on the Intel Pentium® 4 (Willamette) processor.

1 Introduction

One of the characteristics that distinguishes industrial formal verification from that done in academia is that industry often works on large, complex circuits described at the register transfer level (RTL). In comparison, academic work usually verifies either small RTL circuits or high-level abstractions of complex circuits. Academia remains the primary driving force in the development of new verification algorithms, and many of these advances have been successfully transferred from academia to industry [7,6,12,4,15,16,9,2,18]. However, improving the strategies for applying these algorithms in industry requires exposure to circuits of a size and complexity that are rarely available to academics.

An important task in continuing the spread of formal verification in industry is to document and promulgate techniques for applying formal verification tools. In this paper we describe a variety of decomposition strategies that we have used as part of the formal verification project for the Intel Pentium® 4 (Willamette) processor. This paper focuses on strategies taken from verifying two different implementations of queues and a floating-point adder.

The verification tool we use is an LTL model checker developed at Intel that supports a variety of abstraction and decomposition techniques [14,15]. Our strategies should be generally applicable to most model-checking-based verification tools. In theory, the techniques are also applicable for theorem-proving. But, because the capacity limitations of model checking and theorem proving are so very different, a strategy that is effective in reducing the size of a model checking task might not be the best way to reduce the size of the problem for theorem proving.

- hierarchical model checking enables wide variety of decomposition techniques
- techniques:

- symmetry in behavior
 - multiple verifications runs of ``same'' property on different parts of circuit
 - due to optimizations, can have identical behavior (for legal inputs), but different structure
- structural decomposition of circuit and spec
 - propagation of assumptions

2 Overview

In this section we give a high-level view of LTL model checking and provide some intuition about how the model checker works while maintaining our focus on the applications rather than the model checker itself. The focus of this paper is on the application of decomposition techniques, not on the model checking algorithms underlying the decomposition.

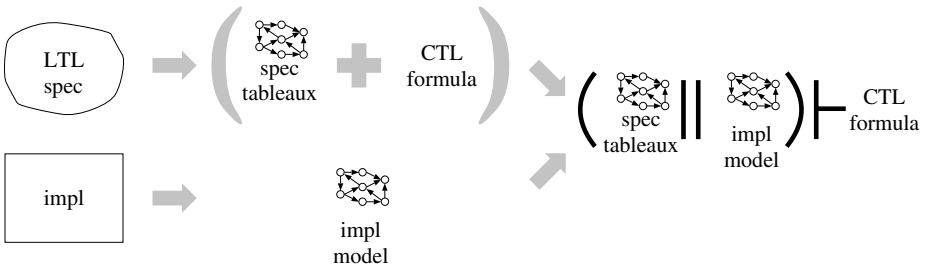
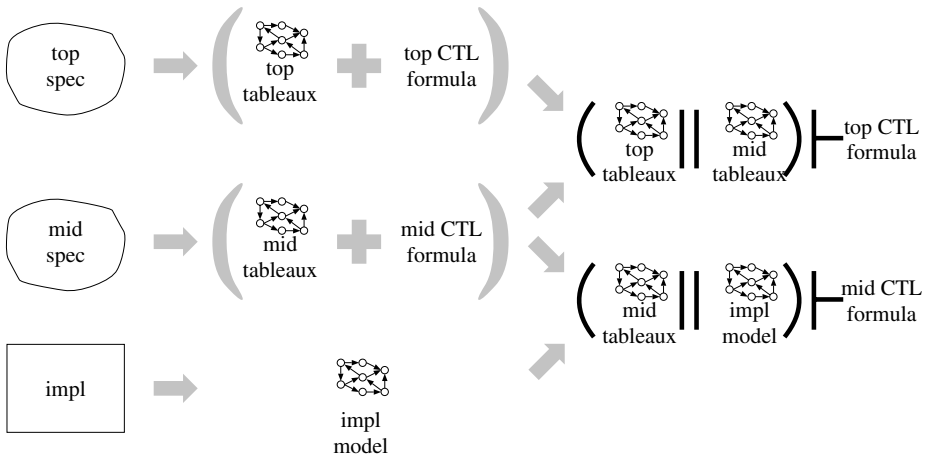
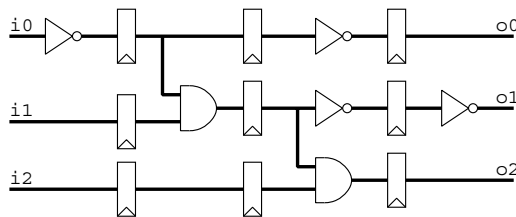


Fig. 1. Verification flow

A high-level view of model checking for a linear temporal logic (LTL) is shown in Figure 1. LTL model checking is done by converting the specification (which is an LTL formula) to a tableaux and a CTL formula. The tableaux is an automaton that recognizes traces that satisfy the LTL formula. The CTL formula checks that the tableaux never fails. The verification run computes the product machine of the tableaux and the implementation and checks that the product machine satisfies the CTL formula [8].

Because specifications are converted into automata, and model checking verifies automata against temporal formulas, specifications can themselves be verified against higher-level specifications. Figure 2 shows an implementation that is verified against a middle-level specification, which, in turn, is verified against a high-level specification.

We now step through a simple example of hierarchical decomposition, using the circuit `pri` shown in Figure 3. The circuit `pri` takes three inputs (`i0`, `i1`, and `i2`) and outputs a 1 on the highest priority output line `line` whose input was a 1. We model combinational logic as having zero delay and flip flops as being unit delay. We draw flip flops as rectangles with small triangles at the bottom.

**Fig. 2.** Hierarchical verification flow**Fig. 3.** Example circuit: pri

Theorem 1 shows an example property that we will verify about `pri`. The property says that, for `pri`, if the output `o0` is 1, then `o2` will be 0. For the sake of illustration, we will pretend that verifying Theorem 1 is beyond the capacity of a model checker, so we decompose the problem. We carry out the verification in three steps by verifying Lemmas 1-3.

Theorem 1. $\text{pri} \models (o0 \implies \neg o2)$

Lemma 1. $\text{pri} \models (o0 \implies \neg o1)$

Lemma 2. $\text{pri} \models (\neg o1 \implies \neg o2)$

Lemma 3. $(\text{Lemma1}, \text{Lemma2}) \models (o0 \implies \neg o2)$

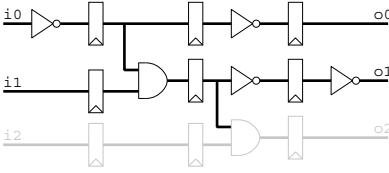


Fig. 4. Cone of influence reduction for Lemma 1

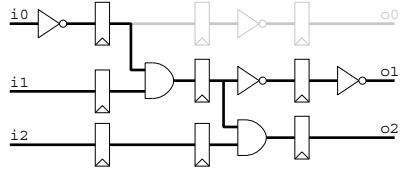


Fig. 5. Cone of influence reduction for Lemma 2

By decomposing the verification, we end up with multiple verification runs, but each one is smaller than Theorem 1, the original run (Table 1). The first two runs (Lemmas 1 and 2) are smaller than Theorem 1 because, using *cone of influence* reduction, Lemmas 1 and 2 do not need to look at the entire circuit. The last run, Lemma 3, glues together the earlier results to verify the top-level specification. When verifying Lemma 3 we do not need the circuit at all, and so its verification is very small and easy to run.

property	latches	inputs	total variables
Theorem 1	9	3	12
Lemma 1	6	2	8
Lemma 2	7	3	10
Lemma 3	0	3	3

Table 1. Summary of verification of `pri`

One advantage of this style of hierarchical verification is that it provides some theorem-proving-like capabilities in a purely model checking environment. Two disadvantages are that the approach is still subject to the expressiveness limitations and some of the capacity limits of BDDs and that it does not support reasoning about the entire chain of reasoning. That is, we use Lemma 3 to verify the right-hand-side of Theorem 1 but we cannot explicitly prove Theorem 1, instead we use a database of specifications to track chains of dependencies.

3 Parallel Ready Queue

This section describes part of the verification of a “parallel ready queue” (`prqueue`) that operates in parallel on k channels (Figure 6). Packets arrive, one at a time, at any of the `arrive` signals. A packet departs the queue if it is the oldest ready packet. Packet i becomes ready when `ready`[i] is 1. Once a packet is ready, it remains ready until it has departed. Packets can become ready to leave the queue in any order. Hence, it is possible for a packet to leave the queue before an older packet if the older packet is not yet ready.

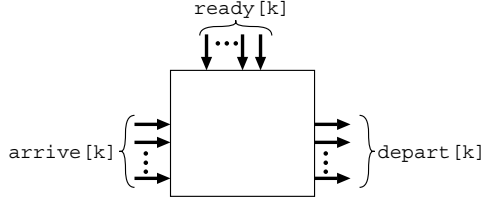


Fig. 6. Parallel queue circuit (`prqueue`)

There are a number of environmental constraints that are required for the circuit to work correctly:

1. A packet will not arrive on a channel if the queue currently contains a packet in that channel.
2. At most one packet will arrive at any time.
3. If a packet arrives on a channel, the corresponding ready signal will eventually be set to 1.
4. A channel will not be marked as ready if it does not contain a packet.

Note, the environment has the freedom to simultaneously mark multiple packets as ready.

The top level specification (Theorem 2) covers both liveness and safety.

Theorem 2. *PriQueue Specification*

1. A packet that arrives on `arrive`[i] will eventually depart on `depart`[i].
2. If a packet departs, it is the oldest ready packet in the queue.

3.1 Implementation

We briefly describe the high-level algorithm for the implementation of `prqueue`. There are many sources of complexity, e.g., datapath power-saving control logic and various gated clock circuits, which we ignore for the sake of simplicity in the following description. The implementation of `prqueue` uses a two-dimensional $k \times k$ scoreboard (`scb`) to maintain the relative age of packets and an array of k `rdy` signals to remember

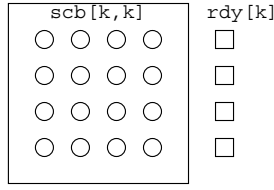


Fig. 7. Scoreboard in parallel ready queue

if a channel is ready. Figure 7 shows the scoreboard and ready array for a queue with four channels.

Initially, all of the elements in the scoreboard and ready array are set to 0. When a packet arrives on `arrive[i]`, all of the scoreboard elements in column `i` are set to 0 and all of the elements in row `i` are set to 1. The intuition behind this algorithm lies in two observations:

- The scoreboard row for the youngest (most recently arrived) packet is the row with the most 1s, while the oldest packet has the row with the most 0s.
- The packet for row `i` is older than the packet for row `j` if and only if $scb[i, j] = 1$.

Please note that the diagonal elements do not contain any semantically useful information. When `ready[i]` is set to 1, packet `i` will be marked as ready to leave the queue. If at least one of the packets is marked as ready, the scoreboard values are examined to determine the relative age of all ready packets and the oldest among them departs the queue. The process of packets arriving, being marked as ready, and departing is illustrated in Figure 8.

As is all too common when working with high-performance circuits, the specification and high-level description of the implementation appear deceptively straightforward. This circuit has been optimized for area and performance and the actual implementation is much more complex than the high-level description. For example, the logic for `ready[i]` signal involves computation of multiple conditions which all must be satisfied before the packet is ready to depart the scoreboard. We have ignored many such details of implementation for the purpose of the simplicity of the description. The actual circuit verified consists of hundreds of latches with significantly larger values for `k`.

3.2 Verification

Theorem 3 is one of the major theorems used in verifying the circuit against the top-level specification 2.

Theorem 3 says: *if a packet is ready, then a packet will depart*. The signal `depValid[i]` is the “valid bit” for `depart[i]`; it says that the packet on row `i` is departing. There are some environmental assumptions such as issues with initialization that we do not encode in this specification for the sake of simplicity.

Theorem 3 (willDepart).

$$(\exists i. rdy[i]) \implies (\exists i. depValid[i])$$

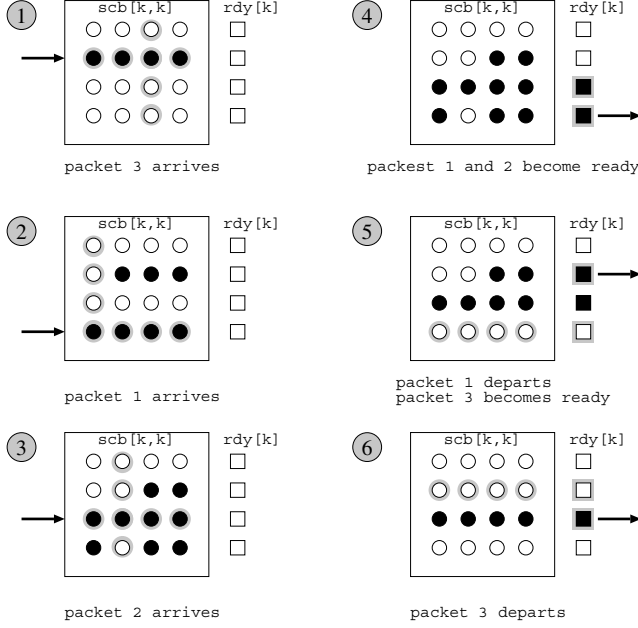


Fig. 8. Scoreboard in parallel ready queue

Theorem 3 is a safety property of the implementation. The logic expression $\exists i. rdy[i]$ stands for the boolean expression $rdy[0] \vee rdy[1] \vee \dots \vee rdy[k]$. Theorem 3 is dependent upon the fanin cone for all k^2 entries in the scoreboard. The fanin cone of the scoreboard consists of hundreds of latches. This amount of circuitry and the temporal nature of information stored in the scoreboard array puts it well beyond the capacity of model checking for values of k found in realistic designs.

To carry out the verification, we decomposed the task into two parts. First, we verified that the scoreboard and ready array impart a transitive and anti-symmetric relationship that describes the relative ages of packets (Lemmas 4 and 5). Second, we verified that these properties imply Theorem 3.

Lemma 4 (Transitivity $i j k$).

$$rdy[i] \wedge rdy[j] \wedge rdy[k] \wedge scb[i, j] \wedge scb[j, k] \implies scb[i, k]$$

Lemma 4 says that if packet i is older than packet j and packet j is older than packet k , then packet i is older than packet k . (Recall that packet i is older than packet j if $scb[i, j] = 1$). The benefit of Lemma 4 is that we reduce the size of the circuit needed from k^2 scoreboard entries to just three entries. The cost of this reduction is that we had to verify the lemma for all possible combinations of scoreboard indices such that $i \neq j \neq k$, which resulted in $k \times (k-1) \times (k-2)$ verification runs.

Lemma 5 (AntiSym $i j$).

$$rdy[i] \wedge rdy[j] \implies scb[i, j] = \neg scb[j, i]$$

Lemma 5 (anti-symmetry) says that if packet i is older than packet j , then packet j cannot be older than packet i (and vice versa.) It is easy to see why this lemma is true by observing that whenever a packet arrives the corresponding column values are set to 0 while the corresponding row values are set to 1. The anti-symmetry lemma relates only two values of the scoreboard entries. We verified Lemma 5 for all possible combinations of i and j (such that $i \neq j$), which resulted in $k \times (k - 1)$ verification runs.

After verifying Lemmas 4 and 5 we verified Theorem 3 by removing the fanin cone of the scoreboard array while assuming the Removing the fanin cone of the scoreboard array is essentially an abstraction which preserves its pertinent properties using transitivity and anti-symmetry lemmas. This particular abstraction preserves the truth of the safety property under consideration. The transitivity and anti-symmetry lemmas provide just enough restriction on the scoreboard values to establish a relative order of arrival among packets that are ready to exit. Note that the transitivity relationship between three scoreboard entries is sufficient to enforce proper values in the scoreboard array even when more than three packets are ready to depart.

The major sources of complexity in verifying Theorem 3 were:

- the large fanin cone of the scoreboard array, which builds up complex temporal information and,
- the size of the scoreboard array: $k \times k$ entries.

We were able to verify Theorem 3 without this hierarchical decomposition for very small values of k . However, the verification task becomes increasingly more difficult for larger values of k . The hierarchical verification approach overcomes the model checking complexity for two reasons:

- The transitivity and anti-symmetry lemmas relate three and two values of the scoreboard entries. The verification complexity of these lemmas is relatively small and is independent of the size of the scoreboard.
- For the verification of Theorem 3, we were able to remove the fanin cone of the entire scoreboard. We enforce proper values in scoreboard entries by enforcing the transitivity and anti-symmetry safety properties which themselves are not of a temporal nature. However, these lemmas together provide enough information to build an implicit order of arrival of packets that allows the implementation to pick the eldest packet.

4 Floating-Point Adder

We briefly preview the basics of binary floating-point numbers [11] before diving into some of the decompositions used in verifying a floating point adder. A floating-point number is represented as a triple (s, e, m) where, s is a sign bit, e is a bit-vector representing the exponent, and m is a bit-vector representing the mantissa. We have verified the sign, mantisa and exponent for both *true addition* and *true subtraction* (the signs of the operands are the same or differ), but consider only the verification of the mantisa for true addition in this paper.

The real number represented by the triple (s, e, m) is:

$$(-1)^s \cdot 2^{\hat{e}-bias} \cdot \hat{m} \cdot 2^{-n_m+1}$$

where \hat{x} is the unsigned integer encoding by the bit vector x and $bias$ n_m is a format-dependent *exponent bias*. The mantissa has an implicit leading 1, and so it represents the value $1.m$, which is in the range $[1, 2)$.

4.1 Implementation

A simple algorithm to calculate the result mantissa for floating-point addition is shown in Figure 9.

```

input: two normal floating-point numbers  $f_1 = (s_1, e_1, m_1)$  and  $f_2 = (s_2, e_2, m_2)$ 

expdiff  :=  $|\hat{e}_1 - \hat{e}_2|$ ;                                absolute difference of exponents
bigman   := if  $(\hat{e}_1 > \hat{e}_2)$  then  $1.m_1$  else  $1.m_2$ ;        mantissa of larger number
smallman := if  $(\hat{e}_1 > \hat{e}_2)$  then  $1.m_2$  else  $1.m_1$ ;        mantissa of smaller number
alzman   := shift_right(smallman, expdiff);              align smaller mantissa with larger
addman   := bigman + alzman;                             add mantissas
resultman := round(addman);                             round result

```

Fig. 9. Simple floating-point true addition algorithm

A simplified implementation for a floating-point true addition circuit `fpadder` is shown in Figure 10. It follows the basic structure of the algorithm in Figure 9. The actual implementation of the adder that we verified is much more complex than this simple depiction. For example, the actual implementation has more than just one adder and more than one parallel datapaths for the sake of performance.

4.2 Verification

We use a reference model similar to the algorithm depicted in Figure 9 as the specification for floating-point addition. Theorem 4 is our top-level theorem for the mantissa.

Theorem 4 (Mantissa).

f_1 and f_2 are normal $\implies (\text{spec_resultman} = \text{impl_resultman})$

For the purpose of the verification, we essentially removed all state elements by *unlatching* the circuit thus arriving at a combinational model of the circuit. We used our model checker, instead of an equivalence checker, to verify the above property on the combinational model as only the model-checking environment provides support for hierarchical decomposition. Also, for the purpose of simplicity we do not include environmental assumptions in the description of Theorem 4.

The main challenge we faced in verifying Theorem 4 was BDD blow-up due to the combined presence of a shifter and adder. As has been reported by others [52],

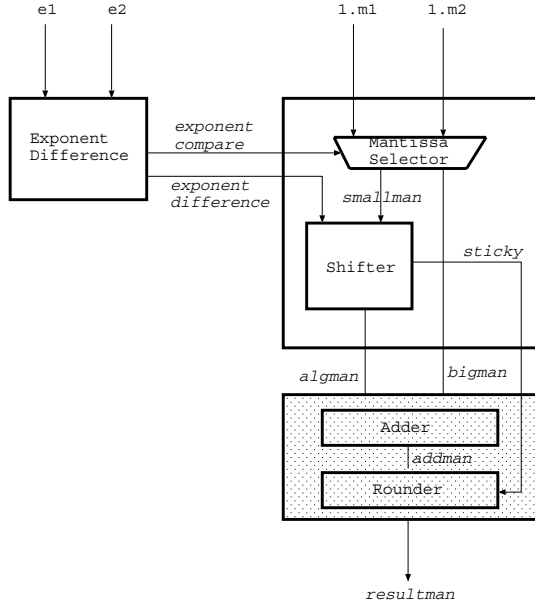


Fig. 10. Simplified circuit for floating-point true addition : fpadder

good variable orderings exist for fixed differences of exponents; however, the BDD size for the combination of variable shifting and addition is exponential with respect to the width of the datapath. Note that both the specification and implementation contain this combination, so we could not represent either the complete implementation or the complete specification with BDDs. Our strategy was to do structural decomposition on *both* the specification and implementation.

Lemma 6 (Shiftermatch).

f_1 and f_2 are normal \implies
 $(\text{spec_alzman} = \text{impl_alzman}) \wedge$
 $(\text{spec_bigman} = \text{impl_bigman}) \wedge$
 $\text{good_properties}(\text{spec_alzman}, \text{spec_bigman})$

We decomposed the specification and implementation at the output of the shifter at the *alzman*, *bigman* and *sticky* signals. This decomposed the verification into two steps: from the inputs of the circuit through the shifter, and the adder and the rounder shown in Figure 10. The two steps are shown in different shades. Through judicious use of cone of influence reductions and other optimizations to both the specification and implementation, we were able to verify the adder and the rounder without needing to include the shifter.

For the first half of the circuit, we verified that the output of the shifter in the specification is equivalent to that in the implementation. Our initial specification for the rounder was that for equal inputs, the specification and the implementation return equal

outputs. However, we discovered that the implementation was optimized such that it returned correct results only for legal combinations of exponent differences and pre-rounded mantissas.

We then began a trial and error effort to derive a set of properties about the difference between the exponents and the pre-rounded mantissa. We eventually identified close to a dozen properties that, taken together, sufficiently constrained the inputs to the rounder such that the implementation matched the specification. Some of the properties are implementation dependent and some are of a more general nature. A few examples of implementation independent properties are:

- The leftmost exponent-difference number of bits of `spec_algman` must be all 0s.
- For exponent differences larger than a specific value, the `sticky` bit and the disjunction of a fixed implementation-dependent number of least significant bits in `spec_algman` must be 1.
- The `expdiff + 1`th leftmost bit of `spec_algman` must be 1.

We modified the original lemma for the first half of the circuit to include these properties. We were able to verify Theorem 4 after removing both the shifters and enforcing the lemmas strengthened with good properties for the shifter and sticky bit.

The above approach illustrates how similar hierarchies present in the implementation and specification can be effectively used to decompose a verification task into smaller and more manageable sub-tasks. We have observed that this technique is particularly applicable when verifying an implementation against a reference model.

In [5], the authors present an approach for verification of floating-point adders based on word-level SMV and multiplicative power HDDs (*PHDDs). In their approach, the specifications of FP adders are divided into hundreds of implementation-independent sub-specifications based on case analysis using the exponent difference. They introduced a technique called short-circuiting to handle the complexity issues arising due to ordering problems.

Our approach uses a model-checker based on BDDs as opposed to word-level model-checker or *PHDDs. Our approach involves a few number of decompositions as opposed to hundreds of cases. However in our approach, the decomposition specifications are implementation-dependent. We believe that for different implementations one can use the same basic approach towards decomposition. The *good_properties* will require the user to incorporate implementation-dependent details. Also, the hierarchical decomposition separates the shifter and the adder and hence does not require any special techniques such as short-circuiting to handle the verification complexity due to conflicting orders requirement.

5 Memory Arrays

Modern microprocessor designs contain dozens of first-in first-out (FIFO) queues throughout their architectures. Certainly, the functional correctness of the processors depends upon the correct behavior of every FIFO in their designs. Due to the evolution of processors (further pipelining, wider datapaths, and so on), verifying just one typical FIFO

can be a challenging task, especially for large FIFOs that are implemented as memory-arrays for area/performance optimizations. Verification of such FIFOs requires decomposing the problem into invariants about the control and datapath implementations. Our work does not refute this notion. Yet, although the proof structure of every FIFO we examined always required many invariants at various level of abstraction, we were able to leverage commonalities in the decomposition paths and methods, even for FIFOs being used in entirely different manners and being verified by different individuals.

5.1 Implementations of Memory Arrays

Memory arrays used as FIFOs are implemented as the array combined with a set of finite state machines that control a write (tail) pointer and a read (head) pointer (Figure 11). Each pointer advances downward with its respective operation, incrementing by the number of elements written to or read from the array, and “wrapping” to the first (top) element when leaving the last (bottom) element. These pointers may take various forms depending upon the physical layout of the memory—linear arrays require only addresses whereas two-dimensional arrays may require an address and an offset into it, which may be implemented as slices of a single address.

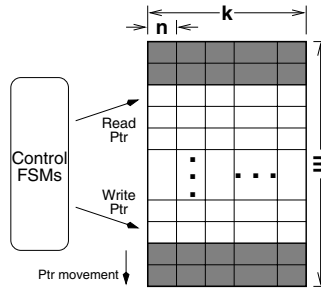


Fig. 11. Memory array circuit

Optimizations for area and performance usually lead to two-dimensional arrays rather than one-dimensional arrays. We use the term *slot* as the storage elements necessary for one queue element, and *row* as a group of slots associated with one address. Thus, an array consists of m rows with k slots per row containing n bits of data.

In one of the arrays we verified, all operations (writes and reads) access an entire row at a time. Other arrays allowed *spanning* writes and/or *partial* reads and writes (see Figure 12). A spanning operation accesses slots in adjacent rows at the same time. A partial operation accesses fewer than k slots, and may or may not be spanning. In a partial access, the offset always changes but the row pointer may not, and vice versa for a spanning operation.

Optimizations in the control FSMs also lead to simplifications that tend to be problematic for verification and decomposition efforts. For example, the FSMs rarely, if ever, take into consideration the position of both pointers. Thus, when examining only

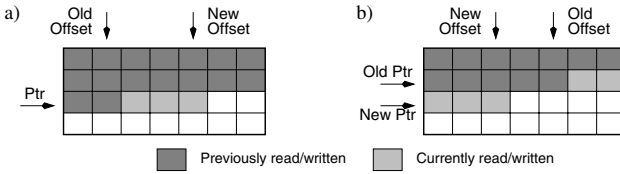


Fig. 12. a) Partial operation, b) Spanning (and partial) operation

the read pointer logic there is nothing to prevent the read pointer from passing the write pointer (an unacceptable behavior). In the absence of constraint properties, only by including the logic of both pointers *and* the datapath could we prevent this from occurring. This detail alone nearly yields memory-array verification intractable for model checkers.

5.2 Overview of Memory Array Verification

Every FIFO in a processor has one or more correctness invariants that the design must satisfy. Most top-level invariants require expected behaviors of the entire control and datapath circuits to verify them. One array we worked with contained over 2100 variables in the necessary logic. Therefore, decomposition was an absolute necessity.

Our technique for verifying memory-array invariants is summarized as follows:

1. Divide the memory array into symmetric, tool-manageable *slices* (e.g., slice equals one or more slots per row, or slice equals one or more rows.)
2. Define a set of properties applicable to every slice.
3. Define properties about the control FSMs that are as abstract as possible (for cone reduction) yet strong enough to guarantee the top-level invariant.
4. Verify the top-level invariants using the properties from 2 and 3.
5. For each slice, verify that if the properties about the control FSMs and the slice hold in a state, then the slice properties will hold in the next state.
6. For each property about the control FSMs, verify that if all properties about the control FSMs hold in a state, then the property will hold in the next state.
7. For any slice or control FSM property that is still intractable with the tools, repeat the decomposition procedure.

We identified a set of five invariants upon the read and write pointers that were used in nearly every memory array abstraction:

- The write pointer changes only when a write occurs.
- The read pointer changes only when a read occurs.
- If a row pointer changes, it increments by one (or more, depending upon the arrays input and output widths) and wraps at the maximum row.
- The read pointer will not pass the write pointer (begin reading bogus data)
- The write pointer will not pass the read pointer (overwrite existing data)

Arrays that allow partial and spanning operations contain offsets indicating which slots in a row are being accessed. Interactions between the offset circuitry and the memory array elements require additional invariants:

- If a read (write) occurs and the row pointer does not change, then the offset increments.
- If a read (write) occurs and the row pointer changes, then the offset decrements or stays the same.

With modern processor designs, teducing the complex nature of the circuitry controlling most FIFOs requires several hierarchical steps just to get to the abstractions above. These levels of the proof trees deal with implementation details for each particular FIFO.

The (deeply) pipelined nature of the array’s datapath and control often requires temporally shifting the effects of these properties in order to substantially reduce the variable counts in “higher” proofs. In other words, using assumptions upon signals in different pipeline stages incurs a penalty if the signals must propagate to further pipestages. It is better to use existing assumptions, which cover the “wrong” pipestages, to prove similar specifications about the propagated signals, especially as the level of abstraction (and freedom given to the model checker) increases. FSM and datapath sizes and design complexities often required temporally shifting the same property to several different times to align with the various FIFO processing stages.

5.3 Array Examples

Here we present two examples of the memory-array verifications we carried out.

Our first example is a “write-once” invariant. This property requires that every slot is written to only once (to prevent destruction) until the read pointer passes it. The FIFO allowed for spanning and partial writes. Thus, every slot on every row has its own write enable line that is calculated from the write pointer, next offset, and global write enable.

For this invariant we sliced the array by row so that the final proofs involved m predicate variables. The predicate variables were defined as sticky bits that asserted if the write enable bits on the row behaved incorrectly. In this “write-once” context, incorrect behavior of the write enables includes the following:

1. The row is not being written to but one or more of the write enables asserted.
2. During a write to the row, a write enable for a slot behind the write pointer (and offset) and ahead of the read pointer (and offset) was asserted.

Note that there are other aberrant behaviors of the write enables but they fall outside the scope of the top-level invariant. They could be included in the predicate variable definitions without affecting the correctness of the final proof. However, needlessly extending the definitions runs the risk of undoing the necessary cone reductions.

The complexity in this verification comes from the inclusion of the read pointer in the predicate definitions. As mentioned before, the control FSMs generally do not take into account the behavior or values of the other pointers or FSMs. This FIFO’s

environmental requirement is that it will never receive more items than it can hold; otherwise, the write pointer could pass the read pointer. Even with this constraint we were forced to fully abstract the read and write pointers before attempting to prove that the predicate bits would never assert.

The last step was to sufficiently abstract the behavior of the predicate variables so they could all be used in the top-level proof. The following invariants were sufficient:

1. Initially, the predicate bit is not asserted.
2. If a row is not being written, its predicate bit does not change.
3. If a row is being written, its predicate bit does not transition from 0 to 1.

These invariants reduce the behavior of the predicate bits to a relation with the write pointer. Together they clearly satisfy the top-level specification (no predicate bit will ever become asserted, thus all write enables behaved as expected) and it seems folly to have to prove them. Nonetheless, the size of the memory array and the capacity limitations of model checking deemed them necessary.

Our second example was our first full-scale attempt at verifying an invariant upon an entire memory array. In this FIFO, control information needed by the read pointer's FSMs is written into the array by the write FSMs. The FIFO's correctness depends upon it being able to write valid and correct control data into the array and then safely and accurately read the data out.

Ideally we would have liked to have been able to prove the correctness of the read FSMs while allowing the control information to freely be any value of the model checker's choosing. However, it quickly became apparent that circuit optimizations required constraining the control data to a "*consistent*" set of legal values. Because rigorous verification work does not particularly enjoy the idea of correct data appearing for free, we were forced to show that control information read from the array always belonged to this *consistent* set. Fortunately, the idea of a valid set of bit values lends itself well to the notion of predicate variables.

The number of bits in the control data set required slicing the array twice: once by row, and then by slot per row. The slot-wise predicate bits indicated whether the slot's current control data was consistent. The row-wise bits, of which there were actually two sets, examined the predicate bits for the entire row while considering the write pointer and offset (or the read pointer and offset). Because the control information flows through the entire FIFO, we had to prove consistency in the following progression through the array's datapath:

- At the inputs to the memory array, for every slot that will be written, the control data is consistent.
- For each array row, if the write pointer moves onto or stays on the row, control data in each slot behind the (new) write offset is consistent.
- For each row, if the write pointer moves off the row, the entire row is consistent.
- For each row, if the write pointer neither moves onto nor moves off of the row, the consistency of each slot's control data does not change.
- For each row, if the read pointer is on the row, and if the write pointer is on the row, then all slots behind the write offset are consistent.

- For each row, if only the read pointer is on the row, all slots are consistent.
- At the memory array’s output muxes, when a read occurs, the *active* output slots contain consistent control data, where *active* is defined by the write and read pointers and offsets (i.e., if pointers on same row, active is between the offsets; otherwise active is above read offset.)

The last step establishes that we will always received consistent control data from the memory array.

As with the FIFO of the write-once invariant, we spent a significant effort reducing the read and write FSM logic to the minimum required behaviors and proving these behaviors at various stages in the pipeline (to align with the FIFO stages described above).

6 Conclusion

Related work in decomposition and model checking can be grouped into three large categories: sound combinations of model checking and theorem proving, ad-hoc combinations of different tools, and hard-coding inference rules into a model checker.

Sound and effective combinations of theorem proving and model checking has been a goal for almost ten years. Joyce and Seger experimented with using trajectory evaluation as a decision procedure for the HOL proof system [13]. They concluded that for hardware verification, most of the user’s interaction is with the model checker, not the proof system. Consequently, using a model checker as a decision procedure in a proof system does not result in an effective hardware verification environment. The PVS proof system has included support for BDDs and mu-calculus model checking [17] and, over time, has received improved support for debugging. Aagaard *et al* have described extensions to the Voss verification system that includes a lightweight theorem proving tool tightly connected to the Voss model checking environment [31]. Gordon is experimenting with techniques to provide a low-level and tight connection between BDDs and theorem proving in the HOL proof system [10].

There are two advantages to combining theorem proving and model checking over a pure model checking based approach, such as we have used. First, general purpose logics provide greater expressability than specification languages tailored to model checking. Second, the sound integration of model checking and theorem proving allows more rigorous results. The principal advantage of the approach outlined here was pragmatic: it enabled us to achieve a more significant result with less effort. Hierarchical model checking allowed an existing group of model checking users to extend the size and quality of their verifications with relatively minor costs in education.

Because finding an effective and sound combination of theorem proving and model checking has been so difficult, a variety of ad hoc combinations have been used to achieve effective solutions at the expense of mechanically guaranteed soundness. Representative of these efforts is separate work by Camilleri and Jang *et al*. Camilleri [4] has used both theorem-proving and model-checking tools in verifying properties of a cache-coherency algorithm. Jang *et al* [12] used CTL model checking to verify a collection of 76 properties about an embedded microcontroller and informal arguments to

convince themselves that their collection of properties were sufficient to claim that they verified their high-level specification.

In an effort to gain some of the capacity improvements provided by a proof system without sacrificing the automation of model checking, McMillan has added inference rules for refinement and decomposition to the Cadence Berkely Labs SMV (CBL SMV) model checker [16]. Eiríksson has used CBL SMV to refine a high-level model of a protocol circuit to a pipelined implementation [9].

The model checker we use shares characteristics of ad hoc combinations of techniques and of adding theorem proving capabilities to a model checker. In essence, the model checker provides us with a decision procedure for propositional logic. This allows us to stay within a purely model checking environment and prove propositional formulae that can be solved by BDDs. However, because we cannot reason about the syntax of formulas, we cannot do Modus Ponens reasoning on arbitrary chains of formulas. Hence, the complexity of our decompositions is limited by the size of BDDs that we can build and we use a separate database facility to ensure the integrity of the overall decomposition.

Although we do not use a theorem prover, we have found that experience with theorem proving can be useful in finding effective decomposition strategies. The challenge is to find *sound* and *effective* decomposition techniques. Assuring the soundness of a decomposition technique benefits from a solid grounding in mathematical logic. Mathematical expertise can also be helpful by providing background knowledge of a menu of decomposition techniques to choose from (e.g., temporal induction, structural induction, data abstraction, etc.). In most situations, many sound decomposition techniques are applicable, but most will not be helpful in mitigating the capacity limitations of model checking. Picking an effective decomposition technique requires knowledge of both the circuit being verified and the model checker being used. We often found that slight modifications to specifications (e.g. the ordering of temporal or Boolean operators) dramatically affect the memory usage or runtime of a verification. Over time, our group developed heuristics for writing specifications so as to extract the maximum capacity from the model checker.

Decomposition techniques similar to the ones that we have described have become standard practice within our group. For this paper we selected illustrative examples to give some insights into how we use decomposition and why we have found it to be successful. Table 2 shows some statistics of a representative sample of the verifications that have used these techniques.

The formal verification effort for the Intel Pentium® 4 processor relied on both formal verification and conventional simulation-based validation. The goal, which in fact was achieved, was that simulation would catch most of the errors and that formal verification would locate pernicious, hard-to-find errors that had gone undetected by simulation.

Acknowledgments

We are indebted to Kent Smith for suggesting the decomposition strategy for the parallel ready queue and many other helpful discussions. We also would like to thank Bob

Circuit	total latches	average number of latches per decomposition	number of verification runs	maximum memory usage	total run time	
t-mem	903	9	70	830MB	27h	
f-r-stall	210	160	10	150MB	23h	
rt-array	500	311	6	110MB	14h	
rq-array	2100	140	300	120MB	100h	Section 5
all-br	1500	175	200	600MB	170h	Section 5
m-buf-com	200	140	20	250MB	20h	
fpadd	2000	400	100	1200MB	2h	Section 4
fpsub	2500	500	100	1800MB	7h	

Table 2. Circuit sizes and verification results

Brennan for the opportunity to perform this work on circuits from Intel microprocessors and Ravi Bulusu for many discussions and helpful comments on the floating-point adder verification.

References

1. M. D. Aagaard, R. B. Jones, K. R. Kohatsu, R. Kaivola, and C.-J. H. Seger. Formal verification of iterative algorithms in microprocessors. In *DAC*, June 2000.
2. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC*, July 1999. (Short paper).
3. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In L. Thery, editor, *Theorem Proving in Higher Order Logics*. Springer Verlag; New York, Sept. 1999.
4. A. Camilleri. A hybrid approach to verifying liveness in a symmetric multi-processor. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, pages 49–68. Springer Verlag; New York, Sept. 1997.
5. Y.-A. Chen and R. Bryant. Verification of floating-point adders. In A. J. Hu and M. Y. Vardi, editors, *CAV*, pages 488–499, July 1998.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Prog. Lang. and Systems*, 16(5):1512–1542, Sept. 1994.
7. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, pages 353–362, 1989.
8. E. M. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press; Cambridge, MA, 1999.
9. A. P. Eiríkson. The formal design of 1m-gate ASICs. In P. Windley and G. Gopalakrishnan, editors, *Formal Methods in CAD*, pages 49–63. Springer Verlag; New York, Nov. 1998.
10. M. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. Technical Report 480, Cambridge Comp. Lab, 1999.
11. IEEE. *IEEE Standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985, 1985.
12. J.-Y. Jang, S. Qadeer, M. Kaufmann, and C. Pixley. Formal verification of FIRE: A case study. In *DAC*, pages 173–177, June 1997.
13. J. Joyce and C.-J. Seger. Linking BDD based symbolic evaluation to interactive theorem proving. In *DAC*, June 1993.

14. G. Kamhi, L. Fix, and O. Weissberg. Automatic datapath extraction for efficient usage of hdds. In O. Grumberg, editor, *CAV*, pages 95–106. Springer Verlag; New York, 1997.
15. S. Mador-Haim and L. Fix. Input elimination and abstraction in model checking. In P. Windley and G. Gopalakrishnan, editors, *Formal Methods in CAD*, pages 304–320. Springer Verlag; New York, Nov. 1998.
16. K. McMillan. Minimalist proof assistants: Interactions of technology and methodology in formal system level verification. In G. C. Gopalakrishnan and P. J. Windley, editors, *Formal Methods in CAD*, page 1. Springer Verlag; New York, Nov. 1998.
17. S. P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Workshop on Industrial-Strength Formal Specification Techniques*, Apr. 1995.
18. Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall. Practical application of formal verification techniques on a frame mux/demux chip from Nortel Semiconductors. In L. Pierre and T. Kropf, editors, *CHARME*, pages 110–124. Springer Verlag; New York, Oct. 1999.

Trends in Computing

Mark E. Dean

IBM Fellow and Vice President, IBM Systems Research
Yorktown Heights, New York, USA
`deanm@us.ibm.com`

Abstract. The ever increasing complexity of computer design, both hardware and software, is moving information technology (IT) companies toward a more disciplined approach for system and component implementation. Design criteria, like time-to-market, zero defects, correct-by-design and high volume manufacturability, are driving engineers to leverage new approaches for hardware and software design. Reusable software components using industry standard component interfaces is one example of how software engineers can quickly implement high-quality code in a short time frame. Formal hardware and software verification tools will also be required to delivery the quality of designs and short design schedules needed to be competitive in the marketplace. But much work is required to develop the tools and design processes that will enable the average engineer to produce complex, correct, reliable and competitive implementations on very short schedules.

The complexity of a modern, commercial computing system almost defies comprehension. At present the complete specification for such systems can only be found in the delivered implementation hardware and software. System complexity is growing as fast as the growth of system performance, circuit density, storage density, network bandwidths and workload diversity. The increasing need to support complex data types (video, audio, 3D images, virtual environments, bio-informatics, etc.) also increases system complexity. There are six industry trends we believe drives the growing complexity of digital technology and of the IT building blocks in general. These trends will pressure engineers to focus on time-to-market, correctness, incremental functionality, and completeness, when implementing competitive system solutions. Engineers, hardware and software, will need to follow a highly disciplined, formal development process, to rapidly deliver accurate implementations of well defined systems.

The six trends driving the industry are:

- **Technology Goes Faster and Faster:** The rate of growth in IT performance and capacities has and will continue to increase at a rapid rate, two times every 12-18 months. This includes high capacity storage, high bandwidth networks, and faster smaller transistors. This trend puts significant pressure on the hardware designer in the face of ever decreasing design schedules and ever increasing expectations in design quality (single pass designs).

- **Everything Goes Online:** New paradigms such as locality-aware devices, always-on mobile connectivity, and environment-aware products, will drive the intermixing of personal and business activities and ever tighter IT integration into human lives. The inter-operability required to satisfy this trend will force the need for industry standards, formal protocol verification, security, and intense collaboration, among companies and institutions.
- **Infrastructure Becomes Intelligent:** Economic incentives increasingly support IT delivery through an information utility paradigm. This paradigm takes advantage of economies of scale and the need for companies to focus on core competencies. The ability to efficiently distribute, mine, store, and create information, will drive infrastructure complexity and intelligence to a level never before believed reasonable or possible.
- **Software Building Blocks:** Higher-level building blocks allow programmers to focus on adding new functionality with more efficiency, quality, flexibility, and speed to market. Software applications and services will be offered as building blocks to higher-level applications and services through the net, supporting new business models. Improvements in code quality and reliability are required to support the demands of 24/7, 99.99% availability.
- **Optimize to Survive:** Sophisticated management and analysis of data will enable winning enterprises to rapidly and flexibly react to market events (sense and respond). Simulation and visualization technologies will allow rapid analysis of data to support business driven decisions.
- **Data Complexity Continues to Increase:** The increased need to leverage complex data types (video, audio, 3D dynamic virtual environments, simulation visualization, etc.) is driving the need to focus on “data-centric” ways of building and programming computer systems. Optimization around a data-centric computing model will require a drastic change in the way we architect, design, and program, computer systems.

These trends will continue to drive drastic change in our industry and provide dynamic and exciting new business opportunities. We must change the way we think about building and programming computer systems to take advantage of the opportunities these trends will provide.

A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD AthlonTM Processor

David M. Russinoff

Advanced Micro Devices, Inc., Austin, TX

Abstract. One obstacle to mathematical verification of industrial hardware designs is that the commercial hardware description languages in which they are usually encoded are too complicated and poorly specified to be readily susceptible to formal analysis. As an alternative to these commercial languages, AMD has developed an RTL language for microprocessor designs that is simple enough to admit a clear semantic definition, providing a basis for formal verification. We describe a mechanical proof system for designs represented in this language, consisting of a translator to the ACL2 logical programming language and a methodology for verifying properties of the resulting programs using the ACL2 prover. As an illustration, we present a proof of IEEE compliance of the floating-point adder of the AMD Athlon processor.

1 Introduction

The formal hardware verification effort at AMD has emphasized theorem proving using ACL2 [4], and has focused on the elementary floating-point operations. One of the challenges of our earlier work was to construct accurate formal models of the targeted circuit designs. These included the division and square root operations of the AMD-K5 processor [1], which were implemented in microcode, and the corresponding circuits of the AMD Athlon processor [2], which were initially modeled in C for the purpose of testing. In both cases, we were required to translate the designs by hand into the logic of ACL2, relying on an unrigorous understanding of the semantics of the source languages.

Ultimately, however, the entire design of the Athlon was specified at the register-transfer level in a hardware description language that was developed specifically for that purpose. Essentially a small synthesizable subset of Verilog with an underlying cycle-based execution model, this language is simple enough to admit a clear semantic definition, providing a basis for formal analysis and verification. Thus, we have developed a scheme for automatically translating RTL code into the ACL2 logic [3], thereby eliminating an important possible

¹ AMD, the AMD logo and combinations thereof, AMD-K5, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

source of error. Using this scheme, we have mechanically verified a number of operations of the Athlon floating-point unit at the register-transfer level, including all addition, subtraction, multiplication, and comparison instructions. As an illustration of our methods, this paper describes the proof of correctness of the Athlon floating-point addition logic, a state-of-the-art adder with leading one prediction logic [1].

Much of the effort involved in the projects mentioned above was in the development and formalization of a general theory of floating-point arithmetic and its bit-level implementation. The resulting ACL2 library [2] is available as a part of ACL2 Version 2.6. Many of the included lemmas are documented in [3] and [4], and some of the details of the formalization are described in [5]. In Sections 2 and 3 below, we present several extensions of the library that were required for the present project.

In Sections 4 and 5, we demonstrate the utility of our floating-point theory, applying it in a rigorous derivation of the correctness of the adder. The theorem reported here is a formulation of the main requirement for IEEE compliance, as stipulated in Standard 754-1985 [6]:

[Addition] shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result ...

In fact, we have also verified that the adder design conforms to other aspects of the behavior prescribed by [6] pertaining to overflow, underflow, and other exceptional conditions, as well as the refinements necessary for Pentium compatibility, as defined in [6]. However, since our main purpose here is to describe a verification methodology rather than to present the details of a specific proof, these secondary results have been omitted from this report.

All of our theorems have been formally encoded as propositions in the logic of ACL2, based on the ACL2 translation of the RTL code, and their proofs have all been mechanically checked with the ACL2 prover. For this purpose, we have developed a proof methodology based on some features of ACL2. In Section 5, we describe both the translation scheme and our proof methodology, using the adder as an illustration.

The use of mechanical theorem proving in the validation of hardware designs is still uncommon in the computer industry, mainly because of the effort that it entails. The work reported here, including the development of the translator and other reusable machinery, consumed some twenty weeks of the author's time, which was divided approximately equally between deriving the informal proofs and checking them mechanically. However, as has been noted before, the cost of formal methods is far outweighed by its potential benefits. In this case, our analysis of the adder exposed a logical error that would have, under certain conditions, resulted in reversing the sign of the sum of zero and an arbitrary nonzero number. This flaw had already survived extensive testing and was unlikely to be detected by conventional validation methods. It was easily repaired in the RTL, but could have been very expensive if not discovered until later.

2 Bit Vectors and Logical Operations

Bit vectors are the fundamental data type of our RTL language as well as the basis of our theory of floating-point arithmetic. We identify the bit vectors of length n with the natural numbers in the range $0 \leq x < 2^n$. Accordingly, we define the k^{th} bit of x to be

$$x[k] = \text{rem}(\lfloor x/2^k \rfloor, 2),$$

while the *slice* of x from the i^{th} bit down through the j^{th} is given by

$$x[i : j] = \lfloor \text{rem}(x, 2^{i+1}) / 2^j \rfloor.$$

The standard binary logical operations, $x \& y$, $x \mid y$, and $x \wedge y$, are defined recursively, e.g.,

$$x \& y = \begin{cases} 0 & \text{if } x = 0 \\ 2(\lfloor x/2 \rfloor \& \lfloor y/2 \rfloor) + 1 & \text{if } x \text{ and } y \text{ are both odd} \\ 2(\lfloor x/2 \rfloor \& \lfloor y/2 \rfloor) & \text{otherwise.} \end{cases}$$

If x is a bit-vector of length n , then its *complement* with respect to n is

$$\text{compl}(x, n) = 2^n - x - 1.$$

Following conventional notation, we shall use the abbreviation

$$\sim x[i : j] = \text{compl}(x[i : j], i - j + 1).$$

The properties of these functions are collected in the ACL2 floating-point library [1]. Many of the basic lemmas are documented in [4] and [5]. Here we shall present several of the more specialized library lemmas that pertain to floating-point addition, in order to illustrate our methods of proof, especially the use of mathematical induction.

The design of the adder involves several computational techniques that are motivated by the observation that while the time required for integer addition increases logarithmically with the inputs, the logical operations defined in Section 2 may be executed in constant time. Thus, for example, the following result, which is readily proved by induction based on the recursive definitions of the logical operations, provides an efficient method for adding three vectors using a two-input adder:

Lemma 1. *For all $x, y, z \in \mathbb{N}$,*

$$x + y + z = (x \wedge y \wedge z) + 2((x \& y) \mid (x \& z) \mid (y \& z)).$$

A more interesting optimization, known as *leading one prediction*, allows the result of a subtraction to be normalized efficiently (in the event of cancelation) by performing the required left shift in advance of the subtraction itself. This requires a prediction of the highest index at which a 1 occurs in the difference.

Although the precise computation of this index appears generally to be as complex as the subtraction itself, a useful approximate solution may be obtained more quickly.

For any $x \in \mathbb{Z}^*$, $\text{expo}(x)$ will denote the index of the leading one of x , i.e., the greatest integer satisfying $2^{\text{expo}(x)} \leq |x|$. Let a and b be integers with $0 < b < a$ and $\text{expo}(a) = e$. We shall compute, in constant time (independent of a and b), a positive integer λ such that $\text{expo}(a - b)$ is either $\text{expo}(a)$ or $\text{expo}(a) - 1$. We begin by defining a function that returns the desired exponent $\phi = \text{expo}(a - b)$. If $\text{expo}(b) < e - 1$, then $a/2 < a - b \leq a$ and we have the trivial solution $\phi = e$. In the remaining case, $e - 1 \leq \text{expo}(b) \leq e$, ϕ may be computed as follows: First, let m be the largest index such that $a[m] > b[m]$, i.e., $a[m] = 1$ and $b[m] = 0$. If $a[m : 0] = 2^m$ and $b[m : 0] = 2^m - 1$, then $\phi = 0$. Otherwise, ϕ is the largest index such that $\phi \leq m$ and $a[\phi - 1] \geq b[\phi - 1]$.

The correctness of this computation is established by the following lemma, in which ϕ is represented as a recursive function:

Lemma 2. *Let $a, b, n \in \mathbb{N}$. For all $d \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $c_k = a[k] - b[k]$ and*

$$\phi(a, b, d, k) = \begin{cases} 0 & \text{if } k = 0 \\ \phi(a, b, c_{k-1}, k-1) & \text{if } k > 0 \text{ and } d = 0 \\ \phi(a, b, d, k-1) & \text{if } k > 0 \text{ and } d \neq 0 \text{ and } d = -c_{k-1} \\ k & \text{if } k > 0 \text{ and } d \neq 0 \text{ and } d \neq -c_{k-1}. \end{cases}$$

If $a < 2^n$, $b < 2^n$, and $a \neq b$, then $\phi(a, b, 0, n) - 1 \leq \text{expo}(a - b) \leq \phi(a, b, 0, n)$.

Proof: It is easy to show, by induction on k , that $\phi(a, b, d, k) = \phi(b, a, -d, k)$. Therefore, we may assume that $a > b$. Note also that if $a[k-1 : 0] = a'[k-1 : 0]$ and $b[k-1 : 0] = b'[k-1 : 0]$, then $\phi(a, b, d, k) = \phi(a', b', d, k)$.

In the case $n = 1$, we have $a = 1$, $b = 0$, and

$$\text{expo}(a - b) = 0 = \phi(a, b, 1, 0) = \phi(a, b, 0, 1).$$

We proceed by induction, assuming $n > 1$.

Suppose first that $c_{n-1} = 0$. Let $a' = a[n-2 : 0]$ and $b' = b[n-2 : 0]$. Then by inductive hypothesis,

$$\phi(a', b', 0, n-1) - 1 \leq \text{expo}(a' - b') \leq \phi(a', b', 0, n-1).$$

But $a - b = a' - b'$, hence $\text{expo}(a - b) = \text{expo}(a' - b')$, and

$$\phi(a, b, 0, n) = \phi(a, b, c_{n-1}, n-1) = \phi(a, b, 0, n-1) = \phi(a', b', 0, n-1).$$

Now suppose that $c_{n-1} = 1$ and $c_{n-2} = -1$. Then $a[n-1] = b[n-2] = 1$ and $a[n-2] = b[n-1] = 0$. It follows that

$$2^{n-1} + 2^{n-2} > a \geq 2^{n-1} > b \geq 2^{n-2}.$$

Let $a' = a - 2^{n-2}$ and $b' = b - 2^{n-2}$. Then

$$2^{n-1} > a' \geq 2^{n-2} > b' \geq 0.$$

Once again,

$$\phi(a', b', 0, n-1) - 1 \leq \text{expo}(a' - b') \leq \phi(a', b', 0, n-1)$$

and $\text{expo}(a - b) = \text{expo}(a' - b')$. But

$$\begin{aligned} \phi(a, b, 0, n) &= \phi(a, b, 1, n-1) = \phi(a, b, 1, n-2) = \phi(a', b', 1, n-2) \\ &= \phi(a', b', 0, n-1). \end{aligned}$$

In the remaining case, $c_{n-1} = 1$ and $c_{n-2} \geq 0$. Now

$$2^n > a \geq a - b \geq 2^{n-1} - b[n-3:0] > 2^{n-1} - 2^{n-2} = 2^{n-2},$$

hence $n-2 \leq \text{expo}(a-b) \leq n-1$, while

$$\phi(a, b, 0, n) = \phi(a, b, 1, n-1) = n-1. \quad \square$$

Thus, we require a general method for computing a number λ such that $\text{expo}(\lambda) = \phi(a, b, 0, e+1)$. First, we handle the relatively simple case $\text{expo}(b) < \text{expo}(a)$:

Lemma 3. *Let $a, b \in \mathbb{N}^*$ with $\text{expo}(b) < \text{expo}(a) = e$, and let*

$$\lambda = 2a[e-1:0] \mid \sim(2b)[e:0].$$

Then $\lambda > 0$ and $\text{expo}(\lambda) - 1 \leq \text{expo}(a-b) \leq \text{expo}(\lambda)$.

Proof: Since

$$\phi(a, b, 0, e+1) = \phi(a, b, 1, e) = \phi(a[e-1:0], b, 1, e),$$

it will suffice to show, according to Lemma 2, that

$$\phi(a[e-1:0], b, 1, e) = \text{expo}(\lambda).$$

Using induction, we shall prove the following more general result: For all $a, b, k \in \mathbb{N}$, if $a < 2^k$ and $b < 2^k$, then

$$\phi(a, b, 1, k) = \text{expo}(2a \mid \sim(2b)[k:0]).$$

If $k = 0$, then $a = b = 0$ and

$$\text{expo}(2a \mid \sim(2b)[k:0]) = \text{expo}(0 \mid 1) = 0 = \phi(a, b, 1, k).$$

Suppose $k > 0$. If $a[k-1] = 0$ and $b[k-1] = 1$, then

$$\begin{aligned} \phi(a, b, 1, k) &= \phi(a, b, 1, k-1) \\ &= \phi(a, b[k-2:0], 1, k-1) \\ &= \phi(a, b - 2^{k-1}, 1, k-1) \\ &= \text{expo}(2a \mid \sim(2b - 2^k)[k-1:0]) \\ &= \text{expo}(2a \mid \sim(2b)[k:0]). \end{aligned}$$

In the remaining case, $\phi(a, b, 1, k) = k$ and either $a[k-1] = 1$ or $b[k-1] = 0$. Since

$$\text{expo}(2a \mid \sim(2b)[k : 0]) = \max(\text{expo}(2a), \text{expo}(\sim(2b)[k : 0])) \leq k,$$

we need only show $\max(\text{expo}(2a), \text{expo}(\sim(2b)[k : 0])) \geq k$. But if $a[k-1] = 1$, then $2a \geq 2 \cdot 2^{k-1} = 2^k$, and if $b[k-1] = 0$, then $b < 2^{k-1}$, which implies $\sim(2b)[k : 0] = 2^{k+1} - 2b - 1 > 2^k - 1$. \square

The next lemma covers the more complicated case $\text{expo}(b) = \text{expo}(a)$:

Lemma 4. *Let $a, b \in \mathbb{N}^*$ such that $a \neq b$ and $\text{expo}(a) = \text{expo}(b) = e > 1$. Let*

$$\begin{aligned}\lambda_t &= a \wedge \sim b[e : 0], \\ \lambda_g &= a \& \sim b[e : 0], \\ \lambda_z &= \sim(a \mid \sim b[e : 0])[e : 0],\end{aligned}$$

$$\begin{aligned}\lambda_0 &= (\lambda_t[e : 2] \& \lambda_g[e - 1 : 1] \& \sim \lambda_z[e - 2 : 0]) \mid \\ &\quad (\sim \lambda_t[e : 2] \& \lambda_z[e - 1 : 1] \& \sim \lambda_z[e - 2 : 0]) \mid \\ &\quad (\lambda_t[e : 2] \& \lambda_z[e - 1 : 1] \& \sim \lambda_g[e - 2 : 0]) \mid \\ &\quad (\sim \lambda_t[e : 2] \& \lambda_g[e - 1 : 1] \& \sim \lambda_g[e - 2 : 0]),\end{aligned}$$

and

$$\lambda = 2\lambda_0 + 1 - \lambda_t[0].$$

Then $\lambda > 0$ and $\text{expo}(\lambda) - 1 \leq \text{expo}(a - b) \leq \text{expo}(\lambda)$.

Proof: Let c_k and ϕ be defined as in Lemma 1. Since $c_e = 0$,

$$\phi(a, b, 0, e + 1) = \phi(a, b, 0, e) = \phi(a, b, c_{e-1}, e - 1),$$

and therefore it will suffice to show that $\lambda \neq 0$ and $\phi(a, b, c_{e-1}, e - 1) = \text{expo}(\lambda)$. In fact, we shall derive the following more general result: For all $n \in \mathbb{N}$, if $n \leq e - 1$ and $a[n : 0] \neq b[n : 0]$, then $\lambda[n : 0] \neq 0$ and

$$\text{expo}(\lambda[n : 0]) = \begin{cases} \phi(a, b, c_n, n) & \text{if } c_n = 0 \text{ or } c_{n+1} = 0 \\ \phi(a, b, -c_n, n) & \text{otherwise.} \end{cases}$$

For the case $n = 0$, note that $a[0] \neq b[0]$ implies $\lambda[0 : 0] = 1$, hence $\text{expo}(\lambda[0 : 0]) = 0$, while $\phi(a, b, c_0, 0) = \phi(a, b, -c_0, 0) = 0$.

We proceed by induction. Let $0 < n \leq e - 1$. Note that for $0 \leq k \leq e - 2$,

$$\begin{aligned}\lambda_0[k] = 1 &\Leftrightarrow \lambda_t[k + 2] = 1 \text{ and } \lambda_g[k + 1] = 1 \text{ and } \lambda_z[k] = 0, \text{ or} \\ &\lambda_t[k + 2] = 0 \text{ and } \lambda_z[k + 1] = 1 \text{ and } \lambda_z[k] = 0, \text{ or} \\ &\lambda_t[k + 2] = 1 \text{ and } \lambda_z[k + 1] = 1 \text{ and } \lambda_g[k] = 0, \text{ or} \\ &\lambda_t[k + 2] = 0 \text{ and } \lambda_g[k + 1] = 1 \text{ and } \lambda_g[k] = 0.\end{aligned}$$

For $0 \leq k \leq e$,

$$\lambda_t[k] = 1 \Leftrightarrow c_k = 0, \quad \lambda_g[k] = 1 \Leftrightarrow c_k = 1, \quad \text{and} \quad \lambda_z[k] = 1 \Leftrightarrow c_k = -1.$$

It follows that for $0 \leq k \leq e - 2$,

$$\begin{aligned} \lambda_0[k] &= 1 \Leftrightarrow c_{k+1} \neq 0, \text{ and} \\ &\quad \text{if } c_{k+2} = 0 \text{ then } c_k \neq -c_{k+1}, \text{ and} \\ &\quad \text{if } c_{k+2} \neq 0 \text{ then } c_k \neq c_{k+1}. \end{aligned}$$

But since $n > 0$, $\lambda[n] = \lambda_0[n - 1]$, and since $n \leq e - 1$,

$$\begin{aligned} \lambda[n] &= 1 \Leftrightarrow c_n \neq 0, \text{ and} \\ &\quad \text{if } c_{n+1} = 0 \text{ then } c_{n-1} \neq -c_n, \text{ and} \\ &\quad \text{if } c_{n+1} \neq 0 \text{ then } c_{n-1} \neq c_n. \end{aligned}$$

If $c_n = 0$, then $\lambda[n] = 0$, hence $\lambda[n : 0] = \lambda[n - 1 : 0] \neq 0$ and

$$\text{expo}(\lambda[n : 0]) = \text{expo}(\lambda[n - 1 : 0]) = \phi(a, b, c_{n-1}, n - 1) = \phi(a, b, c_n, n).$$

Next, suppose $c_n \neq 0$ and $c_{n+1} = 0$. If $c_{n-1} = -c_n$, then $\lambda[n] = 0$, hence $\lambda[n : 0] = \lambda[n - 1 : 0] \neq 0$ and

$$\begin{aligned} \text{expo}(\lambda[n : 0]) &= \text{expo}(\lambda[n - 1 : 0]) = \phi(a, b, -c_{n-1}, n - 1) = \phi(a, b, -c_{n-1}, n) \\ &= \phi(a, b, c_n, n). \end{aligned}$$

But if $c_{n-1} \neq -c_n$, then $\lambda[n] = 1$ and

$$\text{expo}(\lambda[n : 0]) = n = \phi(a, b, c_n, n).$$

Finally, suppose $c_n \neq 0$ and $c_{n+1} \neq 0$. If $c_{n-1} = c_n$, then $\lambda[n] = 0$, $\lambda[n : 0] = \lambda[n - 1 : 0] \neq 0$, and

$$\begin{aligned} \text{expo}(\lambda[n : 0]) &= \text{expo}(\lambda[n - 1 : 0]) = \phi(a, b, -c_{n-1}, n - 1) = \phi(a, b, -c_{n-1}, n) \\ &= \phi(a, b, -c_n, n). \end{aligned}$$

But if $c_{n-1} \neq c_n$, then $\lambda[n] = 1$ and

$$\text{expo}(\lambda[n : 0]) = n = \phi(a, b, -c_n, n). \quad \square$$

Finally, for the purpose of efficient rounding, it will also be useful to predict the *trailing one* of a sum or difference, i.e., the least index at which a 1 occurs. The following lemma provides a method for computing, in constant time, an integer τ that has precisely the same trailing one as the sum or difference of two given operands. As usual, subtraction is implemented through addition, by incrementing the sum of one operand and the complement of the other. Thus, the two cases $c = 0$ and $c = 1$ of the lemma correspond to addition and subtraction, respectively. We omit the proof, which is similar to that of Lemma [4](#).

Lemma 5. *Let $a, b, c, n, k \in \mathbb{N}$ with $a < 2^n$, $b < 2^n$, $k < n$, and $c < 2$. Let*

$$\sigma = \begin{cases} \sim(a \hat{\ } b)[n - 1 : 0] & \text{if } c = 0 \\ a \hat{\ } b & \text{if } c = 1, \end{cases}$$

$$\kappa = \begin{cases} 2(a \mid b) & \text{if } c = 0 \\ 2(a \& b) & \text{if } c = 1, \end{cases}$$

and

$$\tau = \sim(\sigma \hat{\ } \kappa)[n : 0].$$

Then

$$(a + b + c)[k : 0] = 0 \Leftrightarrow \tau[k : 0] = 0.$$

3 Floating Point Numbers and Rounding

Floating point representation of rational numbers is based on the observation that every nonzero rational x admits a unique factorization,

$$x = \text{sgn}(x)\text{sig}(x)2^{\text{expo}(x)},$$

where $\text{sgn}(x) \in \{1, -1\}$ (the *sign* of x), $1 \leq \text{sig}(x) < 2$ (the *significand* of x), and $\text{expo}(x) \in \mathbb{Z}$ (the *exponent* of x).

A floating point representation of x is a bit vector consisting of three fields, corresponding to $\text{sgn}(x)$, $\text{sig}(x)$, and $\text{expo}(x)$. A *floating point format* is a pair of positive integers $\phi = (\sigma, \epsilon)$, representing the number of bits allocated to $\text{sig}(x)$ and $\text{expo}(x)$, respectively. If z is a bit vector of length $\sigma + \epsilon + 1$, then the *sign*, *exponent*, and *significand fields* of z with respect to ϕ are $s = z[\sigma + \epsilon]$, $e = z[\sigma + \epsilon - 1 : \sigma]$, and $m = z[\sigma - 1 : 0]$, respectively. The rational number represented by z is given by

$$\text{decode}(z, \phi) = (-1)^s \cdot m \cdot 2^{e - 2^{\epsilon-1} - \sigma + 2}.$$

If $z[\sigma - 1] = 1$, then z is a *normal ϕ -encoding*. In this case, if $x = \text{decode}(z, \phi)$, then $\text{sgn}(x) = (-1)^s$, $\text{sig}(x) = 2^{\sigma-1}m$, and $\text{expo}(x) = e - (2^{\epsilon-1} - 1)$. Note that the exponent field is biased in order to provide for an exponent range $1 - 2^{\epsilon-1} \leq \text{expo}(x) \leq 2^{\epsilon-1}$.

Let $x \in \mathbb{Q}$ and $n \in \mathbb{N}^*$. Then x is *n-exact* iff $\text{sig}(x)2^{n-1} \in \mathbb{Z}$. It is easily shown that x is *representable with respect to ϕ* , i.e., there exists $z \in \mathbb{N}$ such that $x = \text{decode}(z, \phi)$, iff x is σ -exact and $-2^{\epsilon-1} + 1 \leq \text{expo}(x) \leq 2^{\epsilon-1}$.

The AMD Athlon floating-point unit supports four formats, corresponding to *single*, *double*, and *extended* precision as specified by IEEE, and a larger *internal* format:

$$(24, 7), (53, 10), (64, 15), \text{ and } (68, 18).$$

In our discussion of the adder, floating point numbers will always be represented in the internal (68, 18) format. If z is a bit vector of length 87, then we shall abbreviate $decode(z, (68, 18))$ as \hat{z} .

A *rounding mode* is a function \mathcal{M} that computes an n -exact number $\mathcal{M}(x, n)$ corresponding to an arbitrary rational x and a degree of precision $n \in \mathbb{N}^*$. The most basic rounding mode, *truncation* (round toward 0), is defined by

$$trunc(x, n) = sgn(x) \lfloor 2^{n-1} sig(x) \rfloor 2^{expo(x)-n+1}.$$

Thus, $trunc(x, n)$ is the n -exact number y that is closest to x and satisfies $|y| \leq |x|$. Similarly, rounding *away* from 0 is given by

$$away(x, n) = sgn(x) \lceil 2^{n-1} sig(x) \rceil 2^{expo(x)-n+1},$$

and the three other modes discussed in [4] are defined simply in terms of those two: $inf(x, n)$ (round toward ∞), $minf(x, n)$ (round toward $-\infty$), and $near(x, n)$ (round to the nearest n -exact number, with ambiguities resolved by selecting $(n-1)$ -exact values.

If \mathcal{M} is any rounding mode, $\sigma \in \mathbb{N}^*$, and $x \in \mathbb{Q}$, then we shall write

$$rnd(x, \mathcal{M}, \sigma) = \mathcal{M}(x, \sigma).$$

The modes that are supported by the IEEE standard are *trunc*, *near*, *inf*, and *minf*. We shall refer to these as *IEEE rounding modes*.

As showed in [4], a number can be rounded according to any IEEE rounding mode by adding an appropriate constant and truncating the sum. In particular, if x is a positive integer with $expo(x) = e$, then the *rounding constant* for x corresponding to a given mode \mathcal{M} and degree of precision σ is

$$C(e, \mathcal{M}, \sigma) = \begin{cases} 2^{e-\sigma} & \text{if } \mathcal{M} = near \\ 2^{e-\sigma+1} - 1 & \text{if } \mathcal{M} = inf \\ 0 & \text{if } \mathcal{M} = trunc \text{ or } \mathcal{M} = minf. \end{cases}$$

Lemma 6. *Let \mathcal{M} be an IEEE rounding mode, $\sigma \in \mathbb{Z}$, $\sigma > 1$, and $x \in \mathbb{N}^*$ with $expo(x) \geq \sigma$. Then*

$$rnd(x, \mathcal{M}, \sigma) = trunc(x + C(expo(x), \mathcal{M}, \sigma), \nu),$$

where

$$\nu = \begin{cases} \sigma - 1 & \text{if } \mathcal{M} = near \text{ and } x \text{ is } (\sigma + 1)\text{-exact but not } \sigma\text{-exact} \\ \sigma & \text{otherwise.} \end{cases}$$

An additional rounding mode is critical to the implementation of floating-point addition: If $x \in \mathbb{Q}$, $n \in \mathbb{N}$, and $n > 1$, then

$$sticky(x, n) = \begin{cases} x & \text{if } x \text{ is } (n-1)\text{-exact} \\ trunc(x, n-1) + sgn(x)2^{expo(x)+1-n} & \text{otherwise.} \end{cases}$$

The significance of this operation is that the result of rounding a number x to σ bits, according to any IEEE rounding mode, can always be recovered from $sticky(x, \sigma + 2)$:

Lemma 7. *Let \mathcal{M} be an IEEE rounding mode, $\sigma \in \mathbb{N}^*$, $n \in \mathbb{N}$, and $x \in \mathbb{Q}$. If $n \geq \sigma + 2$, then*

$$\text{rnd}(x, \mathcal{M}, \sigma) = \text{rnd}(\text{sticky}(x, n), \mathcal{M}, \sigma).$$

Proof: We may assume that $x > 0$ and x is not $(n-1)$ -exact; the other cases follow trivially. First, note that since $\text{sticky}(x, n)$ is n -exact but not $(n-1)$ -exact,

$$\begin{aligned} \text{trunc}(\text{sticky}(x, n), n-1) &= \text{sticky}(x, n) - 2^{\text{expo}(\text{sticky}(x, n)) - (n-1)} \\ &= \text{sticky}(x, n) - 2^{\text{expo}(x) + 1 - n} \\ &= \text{trunc}(x, n-1). \end{aligned}$$

Thus, for any $m < n$,

$$\text{trunc}(\text{sticky}(x, n), m) = \text{trunc}(\text{trunc}(x, n-1), m) = \text{trunc}(x, m),$$

and the corresponding result for *away* may be similarly derived.

This disposes of all but the case $\mathcal{M} = \text{near}$. For this last case, it suffices to show that if $\text{trunc}(x, \sigma+1) = \text{trunc}(y, \sigma+1)$ and $\text{away}(x, \sigma+1) = \text{away}(y, \sigma+1)$, then $\text{near}(x, \sigma) = \text{near}(y, \sigma)$. We may assume $x \leq y$. Suppose $\text{near}(x, \sigma) \neq \text{near}(y, \sigma)$. Then for some $(\sigma+1)$ -exact a , $x \leq a \leq y$. But this implies $x = a$, for otherwise $\text{trunc}(x, \sigma+1) \leq x < a \leq \text{trunc}(y, \sigma+1)$. Similarly, $y = a$, for otherwise $\text{away}(x, \sigma+1) \leq a < y \leq \text{away}(y, \sigma+1)$. Thus, $x = y$, a contradiction. \square

The following property is essential for computing a rounded sum or difference:

Lemma 8. *Let $x, y \in \mathbb{Q}$ such that $y \neq 0$ and $x + y \neq 0$. Let $k \in \mathbb{Z}$, $k' = k + \text{expo}(x) - \text{expo}(y)$, and $k'' = k + \text{expo}(x + y) - \text{expo}(y)$. If $k > 1$, $k' > 1$, $k'' > 1$, and x is $(k' - 1)$ -exact, then*

$$x + \text{sticky}(y, k) = \text{sticky}(x + y, k'').$$

Proof: Since x is $(k' - 1)$ -exact, $2^{k-2-\text{expo}(y)}x = 2^{(k'-1)-1-\text{expo}(x)}x \in \mathbb{Z}$. Thus,

$$\begin{aligned} y \text{ is } (k-1)\text{-exact} &\Leftrightarrow 2^{k-2-\text{expo}(y)}y \in \mathbb{Z} \\ &\Leftrightarrow 2^{k-2-\text{expo}(y)}y + 2^{k-2-\text{expo}(y)}x \in \mathbb{Z} \\ &\Leftrightarrow 2^{k''-2-\text{expo}(x+y)}(x+y) \in \mathbb{Z} \\ &\Leftrightarrow x+y \text{ is } (k''-1)\text{-exact}. \end{aligned}$$

If y is $(k-1)$ -exact, then

$$x + \text{sticky}(y, k) = x + y = \text{sticky}(x + y, k'').$$

Thus, we may assume that y is not $(k-1)$ -exact. Now in [\[1\]](#) it was proved, with k , k' , and k'' as defined above, and under the weaker assumptions that $k > 0$, $k' > 0$, $k'' > 0$, and x is k' -exact, that

$$x + \text{trunc}(y, k) = \begin{cases} \text{trunc}(x + y, k'') & \text{if } \text{sgn}(x + y) = \text{sgn}(y) \\ \text{away}(x + y, k'') & \text{if } \text{sgn}(x + y) \neq \text{sgn}(y). \end{cases}$$

Hence, if $\text{sgn}(x + y) = \text{sgn}(y)$, then

$$\begin{aligned} x + \text{sticky}(y, k) &= x + \text{trunc}(y, k - 1) + \text{sgn}(y)2^{\text{expo}(y)+1-k} \\ &= \text{trunc}(x + y, k'' - 1) + \text{sgn}(x + y)2^{\text{expo}(x+y)+1-k''} \\ &= \text{sticky}(x + y, k''). \end{aligned}$$

On the other hand, if $\text{sgn}(x + y) \neq \text{sgn}(y)$, then

$$\begin{aligned} x + \text{sticky}(y, k) &= x + \text{trunc}(y, k - 1) + \text{sgn}(y)2^{\text{expo}(y)+1-k} \\ &= \text{away}(x + y, k'' - 1) - \text{sgn}(x + y)2^{\text{expo}(x+y)+1-k''} \\ &= \text{trunc}(x + y, k'' - 1) + \text{sgn}(x + y)2^{\text{expo}(x+y)+1-k''} \\ &= \text{sticky}(x + y, k''). \quad \square \end{aligned}$$

4 Description of the Adder

A simplified version of the Athlon floating-point adder is represented in the AMD RTL language as the circuit description \mathcal{A} , displayed in Figs. 1–6. As defined precisely in [1], a program in this language consists mainly of *input declarations*, *combinational assignments*, and *sequential assignments*, which have the forms

$$\text{input } s[k : 0]; \quad (1)$$

$$s[k : 0] = E; \quad (2)$$

and

$$s[k : 0] \leq E; \quad (3)$$

respectively, where $k \in \mathbb{N}$, s is a *signal* representing a bit vector of length $k + 1$, and E is an expression constructed from signals and standard logical connectives. Each signal s occurring anywhere in a description must appear in exactly one of the three contexts (1), (2), and (3), and is called an *input*, a *wire*, or a *register*, accordingly. Any signal may also occur in an *output declaration*,

$$\text{output } s[k : 0]; \quad (4)$$

and is then also called an *output*. Note that the circuit \mathcal{A} has five inputs, **a**, **b**, **op**, **rc**, **pc** (Fig. 1), and one output, **r** (Fig. 2), which happens to be a wire (Fig. 6).

A circuit description may also contain *constant definitions* of the form

$$\text{'define } r \ C$$

where r is an identifier and C is either a numerical constant or a pattern representing a set of constants. generalized constant expression. For example, according to the definition of FSUB0 (Fig. 1), the value computed for the assignment

statement to `sub` (Fig. 2) is 1 whenever the value of the 11-bit vector `op` matches the string `1100000x10x`.

Any signal that occurs in the defining expression E for a (non-input) signal s is called a *direct supporter* of s . If s is a wire and s' is any signal, then s *depends* on s' iff s' is a direct supporter either of s or of some wire on which s depends. It is a syntactic requirement of the language that no wire depends on itself.

A *combinational* circuit is one that is free of registers. The semantics of a combinational circuit are particularly simple: the behavior of each output may be described as a function of the inputs. In fact, the same is true of a more general class of circuits, which we define as follows: A circuit description is an *n-cycle simple pipeline* if each of its signals s may be assigned a *cycle number* $\psi(s) \in \{1, \dots, n\}$ such that

- (1) if s is an input, then $\psi(s) = 1$;
- (2) if s is a wire, then $\psi(s') = \psi(s)$ for each direct supporter s' of s ;
- (3) if s is a register, then $\psi(s') = \psi(s) - 1$ for each direct supporter s' of s ;
- (4) if s is an output, then $\psi(s) = n$.

In [1], we present a general semantic definition of the RTL language, associating a function with each signal. This function returns a sequence of values, interpreted as the values assumed by the signal on successive cycles, for a given set of sequences of values of the input signals. It is shown that for a simple pipeline, the value of each signal s on cycle $\psi(s)$ is determined by the values of the inputs on cycle 1. Moreover, this functional dependence on inputs is the same as for the combinational circuit that results from collapsing the pipeline by replacing each sequential assignment (3) by the corresponding combinational assignment (2).

The actual RTL model of the AMD Athlon floating-point adder is a 4-cycle simple pipeline. In order to simplify our analysis of the circuit as well as this presentation, the circuit description \mathcal{A} listed below was derived by modifying the original as follows:

- (1) All sequential assignments have been replaced with combinational assignments, yielding an equivalent combinational circuit.
- (2) All code pertaining to functions other than addition and subtraction, which are performed by the same hardware, has been deleted.
- (3) All code pertaining to the reporting of exceptional conditions, including overflow and underflow, has been deleted.
- (4) The remaining code has been simplified by eliminating signals and combining assignments when possible.
- (5) Signal names have been changed to promote readability.

The resulting circuit \mathcal{A} is shorter and simpler than the original, and bears less resemblance to the intended gate-level implementation, but the two may easily be shown to be semantically equivalent with respect to the computation of sums and differences. In fact, this equivalence has been established mechanically as discussed in Section 4.

```

module A;

//*****
// Definitions
//*****

// CLASS DEFINITIONS
'define UNSUPPORTED      3'b000
'define SNAN              3'b001
'define NORMAL            3'b010
'define INFINITY          3'b011
'define ZERO              3'b100
'define QNAN              3'b101
'define DENORM            3'b110
'define MMX               3'b111

//OPCODE DEFINITIONS//
'define FADD               11'b1100xx0x000
'define FADDU              11'b11010000000
'define FSUB0              11'b1100000x10x
'define FSUB1              11'b1100100x10x
'define FSUB2              11'b1100110x10x
'define FSUBU              11'b11010000100
'define FADDT64            11'b11010010001
'define FSUBT64            11'b11010010000
'define FADDT68            11'b11010010010
'define FSUBT68            11'b11010010110

//PRECISION DEFINITIONS//
'define PC_32               2'b00    // single
'define PC_64               2'b10    // double
'define PC_80               2'b11    // extended
'define PC_80R              2'b01    // extended (reserved)

//ROUNDING DEFINITIONS//
'define RC_RN               2'b00    // round to nearest
'define RC_RM               2'b01    // round to minus infinity
'define RC_RP               2'b10    // round to plus infinity
'define RC_RZ               2'b11    // truncate

//*****
// Parameters
//*****

//INPUTS//
input a[89:0];              //first operand
input b[89:0];              //second operand
input op[10:0];             //opcode
input rc[1:0];              //rounding control
input pc[1:0];              //precision control

```

Fig. 1. Circuit \mathcal{A}

Although it is combinational, our listing of \mathcal{A} reflects the adder's 4-cycle structure insofar as its signals are grouped according to their cycle numbers with respect to the original RTL specification, and our analysis will be guided by this organization. As a first step toward understanding the 4-cycle structure, consider the following procedure, which represents a naive approach to floating point addition and subtraction:

- (1) Compare the exponent fields of the summands to determine the right shift necessary to align the significands;

```

//OUTPUT//
output r[89:0];          //sum or difference

//OPERAND FIELDS//
mana[67:0] = a[67:0]; manb[67:0] = b[67:0];    //significand
expa[17:0] = a[85:68]; expb[17:0] = b[85:68];  //exponent
signa = a[86]; signb = b[86];                 //sign
classa[2:0] = a[89:87]; classb[2:0] = b[89:87]; //class
azero = (classa[2:0] == 'ZERO'); bzero = (classb[2:0] == 'ZERO');

//OPERATION//
int_op = (op == 'FADDT68') | (op == 'FSUBT68');
ext_op = (op == 'FADDT64') | (op == 'FSUBT64');
sub = casex(op[10:0])
      'FSUB0','FSUB1','FSUB2','FSUBU','FSUBT68','FSUBT64' : 1'b1;
      default : 1'b0;
endcase;
esub = sub ^ signa ^ signb;    //effective subtraction

//ROUNDING CONTROL//
rc_near = (rc[1:0] == 'RC_RN') | int_op;      // round to nearest
rc_minf = (rc[1:0] == 'RC_RM') & ~int_op;     // round to minus infinity
rc_inf = (rc[1:0] == 'RC_RP') & ~int_op;      // round to plus infinity
rc_trunc = (rc[1:0] == 'RC_RZ') & ~int_op;     // truncate

//PRECISION CONTROL//
pc_32 = (pc == 'PC_32') & ~ext_op & ~int_op;  // single
pc_64 = (pc == 'PC_64') & ~ext_op & ~int_op;  // double
pc_80 = ((pc == 'PC_80') | (pc == 'PC_80R')) & ~int_op | ext_op; // extended
pc_87 = int_op;    // internal

//*****
// First Cycle
//*****

// SELECT CLOSE OR FAR PATH//
diffpos[18:0] = {1'b0,expa[17:0]} + {1'b0,~expb[17:0]} + 19'b1;
diffneg[17:0] = expb[17:0] + ~expa[17:0] + 18'b1;
swap = ~diffpos[18];
expl[17:0] = bzero | (~azero & ~swap) ? expa : expb;
rsa[6:0] = swap ? diffneg[6:0] : diffpos[6:0];
overshift = (swap & (|diffneg[17:7])) | (~swap & (|diffpos[17:7])) |
             (rsa[6] & ((|rsa[5:3]) | (&rsa[2:1])));
far = ~esub | azero | bzero | overshift | (|rsa[6:1]);

// CLOSE PATH//
shift_close = expa[0] ^ expb[0];
swap_close = ~(expa[0] ^ expa[1] ^ expb[1]);
ina_shift_close[68:0] = shift_close ? {1'b0,mana[67:0]} : {mana[67:0],1'b0};
inb_shift_close[68:0] = shift_close ? {1'b0,manb[67:0]} : {manb[67:0],1'b0};

```

Fig. 2. Circuit \mathcal{A} (continued)

- (2) Perform the required right shift on the significand field that corresponds to the lesser exponent;
- (3) Add (or subtract) the aligned significands, together with the appropriate rounding constant;
- (4) Determine the left shift required to normalize the result;
- (5) Perform the left shift and adjust the exponent accordingly;
- (6) Compute the final result by assembling the sign, exponent, and significand fields.


```

ina_swap_close[68:0] = (shift_close & swap_close) ?
    {manb[67:0] ,1'b0} : {mana[67:0] ,1'b0};
inb_swap_close[68:0] = (shift_close & swap_close) ?
    ina_shift_close[68:0] : inb_shift_close[68:0];
lop0[68:0] = {mana[66:0],2'b0} | {1'b0,~manb[66:0],1'b1};
lop1_t[67:0] = mana[67:0] ^ ~manb[67:0];
lop1_g[67:0] = mana[67:0] & ~manb[67:0];
lop1_z[67:0] = ~(mana[67:0] | ~manb[67:0]);
lop1[67:0] = {1'b0,
    (lop1_t[67:2] & lop1_g[66:1] & ~lop1_z[65:0]) |
    (~lop1_t[67:2] & lop1_z[66:1] & ~lop1_z[65:0]) |
    (lop1_t[67:2] & lop1_z[66:1] & ~lop1_g[65:0]) |
    (~lop1_t[67:2] & lop1_g[66:1] & ~lop1_g[65:0]),
    ~lop1_t[0]};
lop2[68:0] = {manb[66:0],2'b0} | {1'b0,~mana[66:0],1'b1};
lop[68:0] = shift_close ? (swap_close ? lop2[68:0] : lop0[68:0]) : {lop1[67:0],1'b0};
found = 1'b0;
for (i=68; i>=0; i=i-1)
    if (lop[i] & ~found)
        begin
            found = 1'b1;
            lsa[6:0] = 7'h44 - i[6:0];
        end

//FAR PATH//
rshiftin_far[67:0] = swap ? mana[67:0] : manb[67:0];
ina_far[67:0] = azero | (swap & ~bzero) ? manb[67:0] : mana[67:0];

//*****
// Second Cycle
//*****

//PREDICT EXPONENT OF RESULT//
lshift[17:0] = far ? (esub ? 18'h3ffff : 18'b0) : ~{11'b0,lsa[6:0]};
exp[17:0] = expl[17:0] + lshift[17:0];

//ALIGN OPERANDS//
ina_close[68:0] = ~shift_close & (mana < manb) ? inb_swap_close[68:0] << lsa[6:0] :
    ina_swap_close[68:0] << lsa[6:0];
ina_add[70:0] = far ? {ina_far[67:0], 3'b0} : {ina_close[68:0], 2'b0};
inb_close[68:0] = ~shift_close & (mana < manb) ? ina_swap_close[68:0] << lsa[6:0] :
    inb_swap_close[68:0] << lsa[6:0];
rshiftout_far[69:0] = overshift | azero | bzero ?
    70'b0 : {rshiftin_far[67:0],2'b0} >> rsa[6:0];
sticky_t[194:0] = {rshiftin_far[67:0],127'b0} >> rsa[6:0];
sticky_far = ~(azero | bzero) & (overshift | ({sticky_t[124:58]}));
inb_far[70:0] = {rshiftout_far[69:0],sticky_far};
inb_add_nocomp[70:0] = far | azero | bzero ? inb_far[70:0] : {inb_close[68:0],2'b0};
inb_add[70:0] = esub ? ~inb_add_nocomp[70:0] : inb_add_nocomp[70:0];

```

Fig. 3. Circuit \mathcal{A} (continued)

Under the constraints imposed by contemporary technology and microprocessor clock rates, each of the above operations might reasonably correspond to a single cycle, resulting in a six-cycle implementation. It is possible, however, to improve on this cycle count by executing some of these operations in parallel.

The most important optimization of the above algorithm is based on the observation that while a large left shift might be required (in the case of subtraction, if massive cancelation occurs), and a large right shift might be required (if the exponents are vastly different), only one of these possibilities will be re-

```

//DETERMINE SIGN OF RESULT//
sign_tmp = swap | (~far & ~shift_close & (mana < manb)) ? signb ^ sub : signa;
abequal = esub & (mana == manb) & (expa == expb);
sign_reg = ((~azero & ~bzero & ~abequal & sign_tmp) |
             (~azero & ~bzero & abequal & rc_neg) |
             (azero & ~bzero & (signb ^ sub)) |
             (~azero & bzero & signa) |
             (azero & bzero & signa & (signb ^ sub)) |
             (azero & bzero & (signa ^ (signb ^ sub)) & rc_neg)) & ~(ainf | binf) |
             (ainf & signa) | (binf & (signb ^ sub));

//COMPUTE ROUNDING CONSTANT//
int_noco[70:0] = {68'b0,1'b1,2'b0}; // 71'h4
ext_noco[70:0] = case(1'b1)
    rc_trunc : 71'b0;          rc_inf : {64'h0, ~{7 {sign_reg}}};
    rc_near  : {65'b1,6'b0};   rc_minf : {64'h0,{7 {sign_reg}}};
endcase;

doub_noco[70:0] = case(1'b1)
    rc_trunc : 71'h0;          rc_inf : {53'h0,~{18 {sign_reg}}};
    rc_near  : {54'b1,17'b0};   rc_minf : {53'h0,{18 {sign_reg}}};
endcase;
sing_noco[70:0] = case(1'b1)
    rc_trunc : 71'h0;          rc_inf : {24'h0,~{47 {sign_reg}}};
    rc_near  : {25'b1,46'b0};   rc_minf : {24'h0,{47 {sign_reg}}};
endcase;

rconst_noco[70:0] = case(1'b1)
    pc_87 : int_noco;    pc_80 : ext_noco;
    pc_64 : doub_noco;   pc_32 : sing_noco;
endcase;

//*****
// Third Cycle
//*****

//CHECK FOR OVERFLOW OR CANCELLATION//
sum[71:0] = {1'b0,ina_add[70:0]} + {1'b0,inb_add[70:0]} + {71'b0,esub};
overflow = sum[71];
ols = ~sum[70];

//COMPUTE SUM ASSUMING NO OVERFLOW OR CANCELLATION, CHECK FOR CARRYOUT//
sum_noco[70:0] = rconst_noco[70:0] ^ ina_add[70:0] ^ inb_add[70:0];
carry_noco[71:0] = {(rconst_noco[70:0] & ina_add[70:0]) |
                   (rconst_noco[70:0] & inb_add[70:0]) |
                   (ina_add[70:0] & inb_add[70:0]),
                   1'b0};
sum71_noco[72:0] = {2'b0,sum_noco[70:0]} + {1'b0,carry_noco[71:0]} + {72'b0,esub};
overflow_noco = sum71_noco[71];

```

Fig. 4. Circuit \mathcal{A} (continued)

alized for any given pair of inputs. Thus, the Athlon adder includes two data paths: on one path, called the *far* path, the right shift is determined and executed; on the other, called the *close* path, the left shift is performed instead. As noted in Section 2, the left shift may be determined in advance of the subtraction. Consequently, steps (4) and (5) may be executed concurrently with steps (1) and (2), respectively, resulting in a four-cycle implementation. In Section 1, we shall examine the code corresponding to each cycle in detail.

```

//COMPUTE SUM ASSUMING OVERFLOW OR CANCELLATION, CHECK FOR CARRYOUT//
rconst_co[70:0] = esub ? {1'b0,rconst_noco[70:1]} : {rconst_noco[69:0],rconst_noco[0]};
sum_co[70:0] = rconst_co[70:0] ^ ina_add[70:0] ^ inb_add[70:0];
carry_co[71:0] = {(rconst_co[70:0] & ina_add[70:0]) |
                  (rconst_co[70:0] & inb_add[70:0]) |
                  (ina_add[70:0] & inb_add[70:0])},
                  1'b0};
sum71_co[72:0] = {2'b0,sum_co[70:0]} + {1'b0,carry_co[71:0]} + {72'b0,esub};
overflow_co = sum71_co[72];
ols_co = ~sum71_co[70];

//COMPUTE STICKY BIT OF SUM FOR EACH OF THREE CASES//
sticksum[47:0] = esub ? ina_add[47:0] ^ inb_add[47:0] : ~(ina_add[47:0] ^ inb_add[47:0]);
stickcarry[47:0] = esub ? {ina_add[46:0] & inb_add[46:0],1'b0} :
                      {ina_add[46:0] | inb_add[46:0],1'b0};
stick[47:0] = ~(sticksum[47:0] ^ stickcarry[47:0]);
sticky_ols = (!stick[44:16] & pc_32) | (!stick[15:5] & (pc_32 | pc_64)) |
              (!stick[4:1] & ~pc_87) | stick[0] ;
sticky_noco = sticky_ols | (stick[45] & pc_32) | (stick[16] & pc_64) |
              (stick[5] & pc_80) | stick[1] ;
sticky_co =  stick_noco | (stick[46] & pc_32) | (stick[17] & pc_64) |
              (stick[6] & pc_80) | stick[2] ;

//*****
// Fourth Cycle
//*****

//COMPUTE SIGNIFICAND//
man_noco[67:0] =
  {sum71_noco[72] | sum71_noco[71] | sum71_noco[70],
   sum71_noco[69:48],
   sum71_noco[47] & ~(~sum71_noco[46] & ~sticky_noco & pc_32 & rc_near),
   sum71_noco[46:19] & {28 {~pc_32}},
   sum71_noco[18] & ~(pc_32 | (~sum71_noco[17] & ~sticky_noco & pc_64 & rc_near)),
   sum71_noco[17:8] & ~{10{pc_32 | pc_64}},
   sum71_noco[7] & ~(pc_32 | pc_64 | (~sum71_noco[6] & ~sticky_noco & pc_80 & rc_near)),
   sum71_noco[6:4] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_noco[3] & ~(pc_32 | pc_64 | pc_80 | (~sum71_noco[2] & ~sticky_noco & rc_near))
  };

man_co[67:0] =
  {sum71_co[72] | sum71_co[71],
   sum71_co[70:49],
   sum71_co[48] & ~(~sum71_co[47] & ~sticky_co & pc_32 & rc_near),
   sum71_co[47:20] & {28 {~pc_32}},
   sum71_co[19] & ~(pc_32 | (~sum71_co[18] & ~sticky_co & pc_64 & rc_near)),
   sum71_co[18:9] & ~{10{pc_32 | pc_64}},
   sum71_co[8] & ~(pc_32 | pc_64 | (~sum71_co[7] & ~sticky_co & pc_80 & rc_near)),
   sum71_co[7:5] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_co[4] & ~(pc_32 | pc_64 | pc_80 | (~sum71_co[3] & ~sticky_co & rc_near))
  };

```

Fig. 5. Circuit \mathcal{A} (continued)

```

man_ols[67:0] =
  {sum71_co[70] | sum71_co[69],
   sum71_co[68:47],
   sum71_co[46] & ~(~sum71_co[45] & ~sticky_ols & pc_32 & rc_near),
   sum71_co[45:18] & {28 {~pc_32}},
   sum71_co[17] & ~(pc_32 | (~sum71_co[16] & ~sticky_ols & pc_64 & rc_near)),
   sum71_co[16:7] & ~{10{pc_32 | pc_64}},
   sum71_co[6] & ~((pc_32 | pc_64) | (~sum71_co[5] & ~sticky_ols & pc_80 & rc_near)),
   sum71_co[5:3] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_co[2] & ~(pc_32 | pc_64 | pc_80 | (~sum71_co[1] & ~sticky_ols & rc_near))
  };

man_reg[67:0] = case(1'b1)
  (~esub & ~overflow) | (esub & ~ols) : man_noco[67:0];
  ~esub & overflow           : man_co[67:0];
  esub & ols                 : man_ols[67:0];
endcase;

//ADJUST EXPONENT://
exp_noco[17:0] = overflow_noco ? exp[17:0] + 18'h1 : exp[17:0];
exp_co[17:0]   = overflow_co   ? exp[17:0] + 18'h2 : exp[17:0] + 18'b1;
exp_noco_sub[17:0] = overflow ~ overflow_noco ?
  exp[17:0] + 18'h2 : exp[17:0] + 18'h1;
exp_ols[17:0]   = ols_co ? exp[17:0] : exp[17:0] + 18'h1;

exp_reg[17:0] = case(1'b1)
  (~esub & ~overflow) : exp_noco[17:0];
  (~esub & overflow)  : exp_co[17:0];
  (esub & ~ols)       : exp_noco_sub[17:0];
  (esub & ols)        : exp_ols[17:0];
endcase;

//DETERMINE CLASS//
class_reg[2:0] = case(1'b1)
  (azero & bzero) | abequal : 'ZERO;
  default              : 'NORMAL;
endcase;

//FINAL RESULT//
r[89:0] = {class_reg[2:0], sign_reg, exp_reg[17:0], man_reg[67:0]};
endmodule

```

Fig. 6. Circuit \mathcal{A} (continued)

In the subsequent discussion, we shall assume a fixed execution of \mathcal{A} determined by a given set of values corresponding to the inputs. We adopt the convention of italicizing the name of each signal to denote its value for these inputs. Thus, \mathbf{r} denotes the output value determined by the inputs \mathbf{a} , \mathbf{b} , \mathbf{op} , \mathbf{rc} , and \mathbf{pc} .

The input values \mathbf{a} and \mathbf{b} are the operands. Each operand is a vector of ninety bits, including a three-bit encoding of its class, along with sign, exponent, and significand fields, according to the AMD Athlon internal (68,18) format. The fields of \mathbf{a} are assigned to \mathbf{classa} , \mathbf{signa} , \mathbf{expa} , and \mathbf{mana} ; those of \mathbf{b} are similarly recorded. While all eight classes are handled by the actual RTL, we consider here only the case $\mathbf{classa} = \mathbf{classb} = \mathbf{NORMAL}$, and assume that \mathbf{a} and \mathbf{b} are normal (68,18)-encodings, i.e., $\mathbf{mana}[67] = \mathbf{manb}[67] = 1$. Further, in order to ensure that all computed exponent fields are representable in the allotted 18 bits (see,

for example, the proof of Lemma [4\(a\)](#), we assume that $expa$ and $expb$ are both in the range from 69 to $2^{18} - 3$.

The operation to be performed is encoded as the input op , which we shall assume to be one of the 11-bit opcodes listed in Figure 1. This opcode may indicate either addition or subtraction, as reflected by the 1-bit signal sub (Figure 2). Let \mathcal{E} denote the exact result of this operation, i.e.,

$$\mathcal{E} = \begin{cases} \hat{a} + \hat{b} & \text{if } sub = 0 \\ \hat{a} - \hat{b} & \text{if } sub = 1. \end{cases}$$

For simplicity, we shall assume that $\mathcal{E} \neq 0$.

Rounding control is determined by rc along with op . According to these two values, exactly one of the bits rc_near , rc_minf , rc_inf , and rc_trunc is 1, as shown in Figure 2. We shall introduce a variable \mathcal{M} , which we define to be corresponding rounding mode. For example, if op is neither FADDT68 nor FSUBT68 and $rc = RC_RZ$, then $rc_trunc = 1$ and $\mathcal{M} = trunc$.

Similarly, pc and op together determine which one of the bits pc_32 , pc_64 , pc_80 , and pc_87 is set. We define σ to be the corresponding degree of precision: 24, 53, 64, or 68, respectively.

The result prescribed by the IEEE standard will be denoted as \mathcal{P} , i.e.,

$$\mathcal{P} = rnd(\mathcal{E}, \mathcal{M}, \sigma).$$

Our goal may now be stated as follows:

Theorem 1. *Suppose that $a[89 : 77]$ and $b[89 : 87]$ are both NORMAL and that $a[86 : 0]$ and $b[86 : 0]$ are normal (68, 18)-encodings such that $69 \leq a[85 : 68] \leq 2^{18} - 3$ and $69 \leq b[85 : 68] \leq 2^{18} - 3$. Assume that $\mathcal{E} \neq 0$. Then $r[89 : 87] = \text{NORMAL}$, $r[86 : 0]$ is a normal (68, 18)-encoding, and $\hat{r} = \mathcal{P}$.*

Before proceeding with the proof of Theorem [1](#), we must note that the circuit description \mathcal{A} contains a construct of the RTL language that was not described in [3](#), namely, the **for** loop (Fig. 3):

```
found = 1'b0;
for (i=68; i>=0; i=i-1)
  if (lop[i] & ~found)
    begin
      found = 1'b1;
      lsa[6:0] = 7'h44 - i[6:0];
    end
```

If an assignment to a signal s occurs within a **for** loop, then its value s is computed by a recursive function determined by the loop. In this example, the recursive function Θ for the signal lsa is defined by

$$\Theta(lsa, found, lop, i) = \begin{cases} lsa & \text{if } i < 0 \\ \Theta(68 - i, 1, i - 1, lop) & \text{if } lop[i] = 1 \text{ and } found = 0 \\ \Theta(lsa, found, i - 1, lop) & \text{otherwise,} \end{cases}$$

and the value of `lsa` is given by

$$lsa = \Theta(0, 0, val_{\mathcal{A}}(\text{lop}, \mathcal{I}, \mathcal{R}), 68).$$

5 Proof of Correctness

The proof of Theorem [1](#) may be simplified by noting that we may restrict our attention to the case $\mathcal{E} > 0$. In order to see this, suppose that we alter the inputs by toggling the sign bits $a[86]$ and $b[86]$ and replacing rc with rc' , where

$$rc' = \begin{cases} \text{RC_RM} & \text{if } rc = \text{RC_RP} \\ \text{RC_RP} & \text{if } rc = \text{RC_RM} \\ rc & \text{otherwise.} \end{cases}$$

It is clear by inspection of the code that the only signals affected are *signa*, *signb*, *sgn_tmp*, and *sgn_reg*, each of which is complemented, and *rc_inf* and *rc_minf*, which are transposed. Consequently, \hat{r} is negated and \mathcal{M} is replaced by \mathcal{M}' , where

$$\mathcal{M}' = \begin{cases} \text{minf} & \text{if } \mathcal{M} = \text{inf} \\ \text{inf} & \text{if } \mathcal{M} = \text{minf} \\ \mathcal{M} & \text{otherwise.} \end{cases}$$

Now, from the simple identity $rnd(-x, \mathcal{M}', \sigma) = -rnd(x, \mathcal{M}, \sigma)$, it follows that if Theorem [1](#) holds under the assumption $\mathcal{E} > 0$, then it holds generally.

The proof proceeds by examining the signals associated with each cycle in succession.

First Cycle

In the initial cycle, the operands are compared, the path is selected, and the left and right shifts are computed for the close and far paths, respectively. We introduce the following notation:

$$\Delta = |expa - expb|,$$

$$\alpha = \begin{cases} \text{mana} & \text{if } expa > expb \vee (expa = expb \wedge (far = 1 \vee \text{mana} \geq \text{manb})) \\ \text{manb} & \text{otherwise,} \end{cases}$$

and

$$\beta = \begin{cases} \text{manb} & \text{if } \alpha = \text{mana} \\ \text{mana} & \text{if } \alpha = \text{manb}. \end{cases}$$

We begin by comparing the exponents of the operands:

Lemma 9. (a) $swap = 1$ iff $expa < expb$; (b) $expl = \max(expa, expb)$.

Proof: Since $\text{diffpos} = \text{expa} + \sim \text{expb}[17 : 0] + 1 = 2^{18} + \text{expa} - \text{expb} < 2^{19}$,

$$\text{swap} = 1 \Leftrightarrow \text{diffpos}[18] = 0 \Leftrightarrow \text{diffpos} < 2^{18} \Leftrightarrow \text{expa} < \text{expb}. \quad \square$$

The path selection is determined by the signal far , which depends on esub and Δ . For the far path, the magnitude of the right shift is given by Δ . We distinguish between the cases $\Delta \geq 70$ and $\Delta < 70$.

Lemma 10.

- (a) $\text{overshift} = 1$ iff $\Delta \geq 70$;
- (b) if $\text{overshift} = 0$, then $\text{rsa} = \Delta$;
- (c) $\text{far} = 0$ iff $\text{esub} = 1$ and $\Delta \leq 1$.

Proof: If $\text{swap} = 0$, then $\text{diffpos}[17 : 0] = \text{diffpos} - 2^{18} = \text{expa} - \text{expb}$, and if $\text{swap} = 1$, then $\text{diffneg}[17 : 0] = \text{diffneg} = \text{expb} - \text{expa}$. Thus, in either case, $\text{rsa} = \Delta[6 : 0] = \text{rem}(\Delta, 128)$. It follows that $\text{overshift} = 1$ iff either $\Delta[17 : 7] = \lfloor \Delta/128 \rfloor \neq 0$ or $\text{rsa} \geq 70$, which implies (a), from which (b) and (c) follow immediately. \square

The next lemma is an immediate consequence of Lemma 10.

Lemma 11. Assume $\text{far} = 1$.

- (a) $\text{ina_far} = \alpha$;
- (b) $\text{rshiftin_far} = \beta$.

For the close path, the ordering of exponents is determined by shift_close and swap_close :

Lemma 12. Assume $\text{far} = 0$.

- (a) $\text{shift_close} = 0$ iff $\text{expa} = \text{expb}$;
- (b) if $\text{shift_close} = 1$, then $\text{swap_close} = 1$ iff $\text{expa} < \text{expb}$;
- (c) $\text{ina_swap_close} = \begin{cases} 2\text{mana} & \text{if } \text{expa} = \text{expb} \\ 2\alpha & \text{if } \text{expa} \neq \text{expb}; \end{cases}$
- (d) $\text{inb_swap_close} = \begin{cases} 2\text{manb} & \text{if } \text{expa} = \text{expb} \\ \beta & \text{if } \text{expa} \neq \text{expb}. \end{cases}$

Proof: (a) is a consequence of Lemma 10. (c) and (d) follow from (a) and (b). To prove (b), first note that

$$\begin{aligned} \text{expa} > \text{expb} &\Leftrightarrow \text{expa} - \text{expb} = 1 \\ &\Leftrightarrow \text{rem}(\text{expa} - \text{expb}, 4) = 1 \\ &\Leftrightarrow \text{rem}(\text{expa}[1 : 0] - \text{expb}[1 : 0], 4) = 1. \end{aligned}$$

Now suppose that $\text{expa}[0] = 0$. Then $\text{expb}[0] = 1$ and

$$\text{swap_close} = 1 \Leftrightarrow \text{expa}[1] = \text{expb}[1] \Leftrightarrow \text{rem}(\text{expa}[1 : 0] - \text{expb}[1 : 0], 4) = 1.$$

On the other hand, if $\text{expa}[0] = 1$, then $\text{expb}[0] = 0$ and

$$\text{swap_close} = 1 \Leftrightarrow \text{expa}[1] \neq \text{expb}[1] \Leftrightarrow \text{rem}(\text{expa}[1 : 0] - \text{expb}[1 : 0], 4) = 1. \quad \square$$

The magnitude of the left shift is determined by the signal lsa :

Lemma 13. *If $far = 0$, then $66 - lsa \leq expo(\alpha - 2^{-\Delta}\beta) \leq 67 - lsa$.*

Proof: As noted in Section 4, $lsa = \Theta(0, 0, lop, 68)$, where

$$\Theta(lsa, found, lop, i) = \begin{cases} lsa & \text{if } i < 0 \\ \Theta(68 - i, 1, i - 1, lop) & \text{if } lop[i] = 1 \text{ and } found = 0 \\ \Theta(lsa, found, i - 1, lop) & \text{otherwise.} \end{cases}$$

It is easily shown by induction on i that if $0 < i \leq 68$ and $0 < lop < 2^{i+1}$, then $\Theta(0, 0, lop, i) = 68 - expo(lop)$. In particular, if $0 < lop < 2^{69}$, then $lsa = 68 - expo(lop)$. Thus, it will suffice to show that $lop > 0$ and that

$$expo(lop) - 2 \leq expo(\alpha - 2^{-\Delta}\beta) \leq expo(lop) - 1.$$

First consider the case $expa > expb$. We may apply Lemma 4, substituting $2mana$, $manb$, and $lop = lop0$ for a , b , and λ , respectively, and conclude that $lop > 0$ and

$$expo(lop) - 1 \leq expo(2mana - manb) \leq expo(lop).$$

But in this case,

$$expo(2mana - manb) = expo(2(\alpha - 2^{-1}\beta)) = expo(\alpha - 2^{-\Delta}\beta) + 1,$$

and the desired inequality follows. The case $expa < expb$ is similar.

Now suppose $expa = expb$. Here we apply Lemma 4 substituting $mana$, $manb$, and $lop1$ for a , b , and λ , which yields $lop1 > 0$ and

$$expo(lop1) - 1 \leq expo(mana - manb) \leq expo(lop1).$$

But

$$expo(mana - manb) = expo(\alpha - \beta) = expo(\alpha - 2^{-\Delta}\beta)$$

and $expo(lop) = expo(2lop1) = expo(lop1) + 1$. \square

Second Cycle

In the next cycle, the sign of the result is computed, its exponent is approximated, and the significand fields of the operands are aligned by performing the appropriate shifts. The results of this alignment are the signals *ina_add* and *inb_add_nocomp*. In the case of subtraction, *inb_add_nocomp* is replaced by its complement.

In all cases, α and β are first padded on the right with three 0's, and β is shifted right according to Δ . On the close path, both inputs are then shifted left as determined by lsa , and the exponent field of the result is predicted as $expl - lsa - 1$:

Lemma 14. Assume $far = 0$.

- (a) $exp = expl - lsa - 1$;
- (b) $ina_add = rem(2^{lsa+3}\alpha, 2^{71})$;
- (c) $inb_add_nocomp = rem(2^{lsa+3-\Delta}\beta, 2^{71})$.

Proof: (a) The constraints on $expa$ and $expb$ ensure that $69 \leq expl \leq 2^{18} - 3$. Since $0 \leq lsa \leq 68$, it follows that $0 \leq expl - lsa - 1 < 2^{18}$, hence

$$exp = rem(expl + 2^{18} - lsa - 1, 2^{18}) = expl - lsa - 1.$$

- (b) It follows from Lemma 13 that $ina_close = rem(2\alpha \cdot 2^{lsa}, 2^{69})$, hence

$$ina_add = 4rem(2\alpha \cdot 2^{lsa}, 2^{69}) = rem(2^{lsa+3}\alpha, 2^{71}).$$

- (c) This is similar to (b). \square

On the far path, β is rounded after it is shifted to the right. The predicted exponent field is $expl$ for addition and $expl - 1$ for subtraction:

Lemma 15. Assume $far = 1$.

- (a) $exp = expl - esub$;
- (b) $ina_add = 8\alpha$;
- (c) $inb_add_nocomp = \begin{cases} sticky(2^{3-\Delta}\beta, 71 - \Delta) & \text{if } \Delta < 70 \\ 1 & \text{if } \Delta \geq 70. \end{cases}$

Proof: (a) and (b) are trivial, as is (c) in the case $\Delta \geq 70$. Suppose $\Delta < 70$. Since $rshiftin_far = \beta$ and $expo(\beta \cdot 2^{2-\Delta}) = 69 - \Delta$,

$$rshiftout_far = \lfloor \beta \cdot 2^{2-\Delta} \rfloor = trunc(\beta \cdot 2^{2-\Delta}, 70 - \Delta)$$

and $sticky_t = \beta \cdot 2^{127-\Delta}$. Thus,

$$\begin{aligned} sticky_far = 0 &\Leftrightarrow (\beta \cdot 2^{127-\Delta})[124 : 58] = 0 \\ &\Leftrightarrow (\beta \cdot 2^{127-\Delta})[124 : 0] = 0 \\ &\Leftrightarrow \beta \cdot 2^{127-\Delta} \text{ is divisible by } 2^{125} \\ &\Leftrightarrow \beta \cdot 2^{2-\Delta} \in \mathbb{Z} \\ &\Leftrightarrow \beta \text{ is } (70 - \Delta)\text{-exact.} \end{aligned}$$

Thus, if $sticky_far = 0$, then

$$\begin{aligned} inb_add_nocomp &= 2 \cdot rshiftout_far + 0 \\ &= trunc(\beta \cdot 2^{3-\Delta}, 70 - \Delta) \\ &= sticky(\beta \cdot 2^{3-\Delta}, 71 - \Delta), \end{aligned}$$

and otherwise,

$$\begin{aligned} inb_add_nocomp &= 2 \cdot rshiftout_far + 1 \\ &= trunc(\beta \cdot 2^{3-\Delta}, 70 - \Delta) + 2^{(3-\Delta+67)+1-(71-\Delta)} \\ &= sticky(\beta \cdot 2^{3-\Delta}, 71 - \Delta). \quad \square \end{aligned}$$

It is clear that the sign of the final result is correctly given by $sign_reg$:

Lemma 16. *If $\mathcal{E} > 0$, then $\text{sign_reg} = 0$.*

The rounding constant is also computed in this cycle. The following lemma is easily verified for all possible values of \mathcal{M} and σ :

Lemma 17. *If $\mathcal{E} > 0$, then $\text{rconst_noco} = \mathcal{C}(70, \mathcal{M}, \sigma)$.*

Third Cycle

In this cycle, three sums are computed in parallel. The first of these is the unrounded sum

$$\text{sum} = \text{ina_add} + \text{inb_add} + \text{esub}.$$

In the case of (effective) addition, the leading bit $\text{sum}[71]$ is examined to check for overflow, i.e., to determine whether $\text{expo}(\text{sum})$ is 70 or 71. In the subtraction case, only $\text{sum}[70 : 0]$ is of interest—the leading bit $\text{sum}[70]$ is checked to determine whether cancelation occurred. Thus, one of three possible rounding constants is required, depending on whether the exponent of the unrounded sum is 69, 70, or 71. The second adder computes the rounded sum assuming an exponent of 70; the third assumes 71 in the addition case and 69 for subtraction. Concurrently, the relevant sticky bit of the unrounded sum is computed for each of the three cases.

Lemma 18. *Assume $\mathcal{E} > 0$ and $\text{esub} = 0$.*

- (a) $\text{expo}(\text{sum}) = \begin{cases} 70 & \text{if } \text{overflow} = 0 \\ 71 & \text{if } \text{overflow} = 1; \end{cases}$
- (b) $\text{rnd}(\text{sum}, \mathcal{M}, \sigma) = 2^{2^{17} + 69 - \text{exp}\mathcal{P}}.$

Proof: In this case,

$$\text{sum} = \text{ina_add} + \text{inb_add}.$$

Suppose $\Delta < 70$. Then by Lemma 1,

$$\begin{aligned} \text{sum} &= 8\alpha + \text{sticky}(2^{3-\Delta}\beta, 71 - \Delta) \\ &= 8\text{sticky}(\alpha + 2^{-\Delta}\beta, \text{expo}(\alpha + 2^{-\Delta}\beta) + 4), \end{aligned}$$

where $67 \leq \text{expo}(\alpha + 2^{-\Delta}\beta) \leq 68$.

On the other hand, if $\Delta \geq 70$, then since $\text{expo}(\alpha) = \text{expo}(\beta) = 67$, we have $0 < 2^{2-\Delta}\beta < 1$ and $\text{expo}(\alpha + 2^{-\Delta}\beta) = 67$, which imply

$$\begin{aligned} \text{trunc}(\alpha + 2^{-\Delta}\beta, 70) &= \lfloor 2^{70-1-67}(\alpha + 2^{-\Delta}\beta) \rfloor 2^{67+1-70} \\ &= \lfloor 2^2\alpha + 2^{2-\Delta}\beta \rfloor 2^{-2} \\ &= \alpha \end{aligned}$$

and

$$\begin{aligned}
 sum &= 8\alpha + 1 \\
 &= 8(trunc(\alpha + 2^{-\Delta}\beta, 70) + 2^{expo(\alpha + 2^{-\Delta}\beta) + 1 - 71}) \\
 &= 8sticky(\alpha + 2^{-\Delta}\beta, 71) \\
 &= 8sticky(\alpha + 2^{-\Delta}\beta, expo(\alpha + 2^{-\Delta}\beta) + 4).
 \end{aligned}$$

In either case, $70 \leq expo(sum) \leq 71$, which yields (a).

To complete the proof of (b), note first that

$$|\hat{a}| + |\hat{b}| = 2^{expa - 2^{17} - 66}mana + 2^{expb - 2^{17} - 66}manb = 2^{expl - 2^{17} - 66}(\alpha + 2^{-\Delta}\beta),$$

and hence, since $exp = expl$,

$$rnd(\alpha + 2^{-\Delta}\beta, \mathcal{M}, \sigma) = 2^{-exp + 2^{17} + 66}rnd(|\hat{a}| + |\hat{b}|, \mathcal{M}, \sigma) = 2^{-exp + 2^{17} + 66}\mathcal{P}.$$

Now since $\sigma \leq 68$, Lemma 1 implies

$$\begin{aligned}
 rnd(sum, \mathcal{M}, \sigma) &= 2^3rnd(sticky(\alpha + 2^{-\Delta}\beta, expo(\alpha + 2^{-\Delta}\beta) + 4), \mathcal{M}, \sigma) \\
 &= 2^3rnd(\alpha + 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\
 &= 2^{2^{17} + 69 - exp}\mathcal{P}. \quad \square
 \end{aligned}$$

Lemma 19. Assume $\mathcal{E} > 0$ and $esub = 1$.

- (a) $expo(sum[70 : 0]) = \begin{cases} 70 & \text{if } ols = 0 \\ 69 & \text{if } ols = 1; \end{cases}$
- (b) $rnd(sum[70 : 0], \mathcal{M}, \sigma) = 2^{2^{17} + 68 - exp}\mathcal{P}$.

Proof: In this case,

$$sum = ina_add + inb_add + 1 = 2^{71} + ina_add - inb_add_nocomp.$$

Also note that

$$||\hat{a}| - |\hat{b}|| = 2^{expl - 2^{17} - 66}(\alpha - 2^{-\Delta}\beta),$$

hence

$$rnd(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) = 2^{-expl + 2^{17} + 66}rnd(||\hat{a}| - |\hat{b}||, \mathcal{M}, \sigma) = 2^{-expl + 2^{17} + 66}\mathcal{P}.$$

Suppose first that $far = 0$. By Lemmas 1 and 2,

$$\begin{aligned}
 sum[70 : 0] &= rem(rem(2^{lsa+3}\alpha, 2^{71}) - rem(2^{lsa+3-\Delta}\beta, 2^{71}), 2^{71}) \\
 &= rem(2^{lsa+3}(\alpha - 2^{-\Delta}\beta), 2^{71}) \\
 &= 2^{lsa+3}(\alpha - 2^{-\Delta}\beta),
 \end{aligned}$$

where $69 \leq \text{expo}(2^{lsa+3}(\alpha - 2^{-\Delta}\beta)) \leq 70$. Thus, since $\text{exp} = \text{expl} - \text{lsa} - 1$,

$$\begin{aligned} \text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma) &= 2^{lsa+3} \text{rnd}(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\ &= 2^{lsa+3} 2^{-\text{expl}+2^{17}+66} \mathcal{P} \\ &= 2^{2^{17}+68-\text{exp}} \mathcal{P}. \end{aligned}$$

Next, suppose $\text{far} = 1$. Then $\Delta \geq 2$, and it follows that $66 \leq \text{expo}(\alpha - 2^{-\Delta}\beta) \leq 67$. If $\Delta < 70$, then

$$\begin{aligned} \text{sum}[70 : 0] &= 8\alpha - \text{sticky}(2^{3-\Delta}\beta, 71 - \Delta) \\ &= 8\text{sticky}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 4). \end{aligned}$$

But if $\Delta \geq 70$, then $0 < 2^{2-\Delta}\beta < 1$, and hence

$$\begin{aligned} \text{trunc}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 3) \\ = \lfloor 2^2(\alpha - 2^{-\Delta}\beta) \rfloor 2^{-2} = \lfloor 2^2\alpha - 2^{2-\Delta}\beta \rfloor 2^{-2} = (2^2\alpha - 1)2^{-2} = \alpha - 2^{-2}, \end{aligned}$$

which implies

$$\begin{aligned} \text{sticky}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 4) \\ = \text{trunc}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 3) + 2^{-3} = \alpha - 2^{-3}, \end{aligned}$$

and again

$$\text{sum}[70 : 0] = 8\alpha - 1 = 8\text{sticky}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 4).$$

Thus,

$$69 \leq \text{expo}(\text{sum}[70 : 0]) = \text{expo}(\alpha - 2^{-\Delta}\beta) + 3 \leq 70.$$

Since

$$\text{expo}(\alpha - 2^{-\Delta}\beta) + 4 \geq 70 \geq \sigma + 2$$

and $\text{exp} = \text{expl} - 1$, Lemma [1](#) implies

$$\begin{aligned} \text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma) &= 2^3 \text{rnd}(\text{sticky}(\alpha - 2^{-\Delta}\beta, \text{expo}(\alpha - 2^{-\Delta}\beta) + 4), \mathcal{M}, \sigma) \\ &= 2^3 \text{rnd}(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\ &= 2^{2^{17}+68-\text{exp}} \mathcal{P}. \quad \square \end{aligned}$$

The next lemma is a straightforward application of Lemma [1](#).

Lemma 20. $\text{sum71_noco} = \text{rconst_noco} + \text{sum}$.

If the exponent of the sum is not 70, a different rounding constant must be used. Applying Lemma [1](#) again, and referring to the definition of \mathcal{C} , we have the following:

Lemma 21. $sum71_co = rconst_co + sum$, where if $\mathcal{E} > 0$,

$$rconst_co = \begin{cases} \mathcal{C}(71, \mathcal{M}, \sigma) & \text{if } esub = 0 \\ \mathcal{C}(69, \mathcal{M}, \sigma) & \text{if } esub = 1. \end{cases}$$

The next lemma is required for the *near* case:

Lemma 22. Let $S = \begin{cases} sum & \text{if } esub = 0 \\ sum[70 : 0] & \text{if } esub = 1. \end{cases}$ Then S is $(\sigma + 1)$ -exact iff any of the following holds:

- (a) $esub = 0$, $overflow = 0$, and $sticky_noco = 0$;
- (b) $esub = 0$, $overflow = 1$, and $sticky_co = 0$;
- (c) $esub = 1$, $ols = 0$, and $sticky_noco = 0$;
- (d) $esub = 1$, $ols = 1$, and $sticky_ols = 0$.

Proof: S is $(\sigma + 1)$ -exact iff S is divisible by $2^{expo(S) - \sigma}$, i.e., $sum[expo(S) - \sigma - 1 : 0] = 0$. Invoking Lemma 1 with $a = ina_add[47 : 0]$, $b = inb_add[47 : 0]$, $c = esub$, and $n = 48$, we conclude that for all $k < 48$, $sum[k : 0] = 0$ iff $stick[k : 0] = 0$. In particular, since $expo(S) - \sigma - 1 \leq 71 - 24 - 1 = 46$, S is $(\sigma + 1)$ -exact iff $stick[expo(S) - \sigma - 1 : 0] = 0$.

Suppose, for example, that $esub = ols = 1$ and $\sigma = 24$. In this case,

$$\begin{aligned} sticky_ols = 0 &\Leftrightarrow stick[44 : 16] = stick[15 : 5] = stick[4 : 1] = stick[0] = 0 \\ &\Leftrightarrow stick[44 : 0] = 0. \end{aligned}$$

But $expo(S) - \sigma - 1 = 69 - 24 - 1 = 44$, hence S is $(\sigma + 1)$ -exact iff $sticky_ols = 0$. All other cases are similar. \square

Fourth Cycle

In the final cycle, the significand field is extracted from the appropriate sum, and the exponent field is adjusted as dictated by overflow or cancelation. The resulting output r is an encoding of the prescribed result \mathcal{P} , as guaranteed by Theorem 1.

We now complete the proof of the theorem. As noted earlier, we may assume that $\mathcal{E} > 0$. By Lemma 2, $sign_reg = 0$, hence our goal is to show that

$$2^{exp_reg - 2^{17} - 66} man_reg = \mathcal{P}.$$

We shall present the proof for the case $esub = 1$, $ols = 0$; the other three cases are similar.

By Lemma 2, $expo(sum[70 : 0]) = 70$ and

$$rnd(sum[70 : 0], \mathcal{M}, \sigma) = 2^{2^{17} + 68 - exp\mathcal{P}}.$$

Since $man_reg = man_noco$ and $exp_reg = exp_noco_sub$, we must show that

$$\begin{aligned} man_noco &= 2^{-exp_noco_sub + 2^{17} + 66} \mathcal{P} \\ &= 2^{-exp_noco_sub + 2^{17} + 66} 2^{-2^{17} - 68 + exp\mathcal{P}} rnd(sum[70 : 0], \mathcal{M}, \sigma) \\ &= 2^{exp - exp_noco_sub - 2} rnd(sum[70 : 0], \mathcal{M}, \sigma). \end{aligned}$$

Let

$$\nu = \begin{cases} \sigma - 1 & \text{if } \mathcal{M} = \text{near} \text{ and } \text{sum}[70 : 0] \text{ is } (\sigma + 1)\text{-exact but not } \sigma\text{-exact} \\ \sigma & \text{otherwise.} \end{cases}$$

By Lemmas [1](#) and [2](#)

$$\text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma) = \text{trunc}(\text{sum}[70 : 0] + \text{rconst_noco}, \nu).$$

By Lemma [3](#), $\text{sum}[70 : 0]$ is $(\sigma + 1)$ -exact iff $\text{sticky_noco} = 0$. If $\text{sticky_noco} = 0$ and $\mathcal{M} = \text{near}$, then

$$\begin{aligned} \text{sum}[70 : 0] \text{ is } \sigma\text{-exact} &\Leftrightarrow \text{sum}[70 - \sigma] = 0 \\ &\Leftrightarrow (\text{sum} + 2^{70-\sigma})[70 - \sigma] = \text{sum71_noco}[70 - \sigma] = 1. \end{aligned}$$

Thus,

$$\nu = \begin{cases} \sigma - 1 & \text{if } \mathcal{M} = \text{near} \text{ and } \text{sticky_noco} = \text{sum71_noco}[70 - \sigma] = 0 \\ \sigma & \text{otherwise,} \end{cases}$$

and it is easy to check, for each possible value of σ , that

$$\text{man_noco}[66 : 0] = \text{sum71_noco}[69 : 71 - \nu] \cdot 2^{68-\nu}.$$

Since $\text{expo}(\text{sum}[70 : 0]) = 70$ and $\text{expo}(\text{rconst_noco}) \leq 70 - \sigma$,

$$70 \leq \text{expo}(\text{sum}[70 : 0] + \text{rconst_noco}) \leq 71,$$

and therefore

$$\begin{aligned} \text{expo}(\text{sum}[70 : 0] + \text{rconst_noco}) = 70 &\Leftrightarrow (\text{sum}[70 : 0] + \text{rconst_noco})[71] = 0 \\ &\Leftrightarrow (\text{sum} + \text{rconst_noco})[71] = \text{sum}[71] \\ &\Leftrightarrow \text{overflow_noco} = \text{overflow}. \end{aligned}$$

Suppose first that $\text{overflow_noco} \neq \text{overflow}$. Since $\text{expo}(\text{sum}[70 : 0]) = 70$ and

$$\text{expo}(\text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma)) = \text{expo}(\text{sum}[70 : 0] + \text{rconst_noco}) = 71,$$

$\text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma) = 2^{71}$. In this case, $\text{exp_noco_sub} = \text{exp} + 2$, so we must prove that

$$\text{man_noco} = 2^{-4} \text{rnd}(\text{sum}[70 : 0], \mathcal{M}, \sigma) = 2^{67}.$$

Since

$$2^{71} \leq \text{sum}[70 : 0] + \text{rconst_noco} < 2^{71} + 2^{71-\sigma},$$

we have

$$\text{sum71_noco}[70 : 0] = (\text{sum}[70 : 0] + \text{rconst_noco})[70 : 0] < 2^{71-\sigma},$$

which implies $sum71_noco[70 : 71 - \sigma] = 0$, and therefore $man_noco[66 : 0] = 0$. But since $2^{70} \leq sum71_noco < 2^{73}$,

$$man_noco[67] = sum71_noco[72] \mid sum71_noco[71] \mid sum71_noco[70] = 1$$

and $man_noco = 2^{67}$.

Now suppose that $overflow_noco = overflow$. Since $exp_reg = exp + 1$, we must show

$$man_noco = 2^{-3}rnd(sum[70 : 0], \mathcal{M}, \sigma).$$

But since $expo(sum[70 : 0] + rconst_noco) = 70$,

$$\begin{aligned} man_noco &= sum71_noco[70 : 71 - \nu] \cdot 2^{68-\nu} \\ &= 2^{-3}(sum71_noco[70 : 0] \& (2^{71} - 2^{71-\nu})) \\ &= 2^{-3}trunc(sum71_noco[70 : 0], \nu) \\ &= 2^{-3}trunc(sum[70 : 0] + rconst_noco, \nu) \\ &= 2^{-3}rnd(sum[70 : 0], \mathcal{M}, \sigma). \quad \square \end{aligned}$$

6 ACL2 Formalization

In this section, we describe formalization of Theorem 1, including the automatic translation of the RTL model to the ACL2 logic, the formal statement of the theorem, and the mechanization of its proof. Naturally, a prerequisite for all of this is the formalization of the general theory of bit vectors and floating-point arithmetic. This is described in some detail in [1], and the reader may also refer to the complete on-line floating-point library [2].

Our translator is an ACL2 program that accepts any simple pipeline (as defined in Section 1) and automatically produces an equivalent set of executable ACL2 function. For the present purpose, the translator was applied to two circuit descriptions. The first was the actual register-transfer logic for the AMD Athlon adder, the main object of interest. The second was a simplified version, an extension of the circuit \mathcal{A} shown in Section 1, including additional inputs and outputs pertaining to overflow, underflow, and other exceptional conditions. The proof of equivalence of the corresponding resulting ACL2 functions was a critical step in verifying the correctness of the adder.

The first task of the translation is to check that the given circuit description is indeed a simple pipeline, so that it may be replaced by an equivalent combinational circuit. An ordering of the signals of the circuit is then constructed, with respect to which each signal is preceded by those on which it depends.

The main enterprise of the translator is the definition of an ACL2 function corresponding to each signal s , excluding inputs, based on the RTL expression for s . This requires a translation from each primitive RTL operation to an appropriate primitive or defined function of ACL2. For example, the function definition generated for the signal `sticksum` of Fig. 4, constructed from the assignment

```

sticksum[47:0] = esub ?
                ina_add[47:0] ^ inb_add[47:0]
                : ~(ina_add[47:0] ^ inb_add[47:0]);

```

is

```

(defun sticksum (inb-add ina-add esub)
  (if (equal esub 0)
      (comp1 (logxor (bits ina-add 47 0)
                     (bits inb-add 47 0))
            48)
      (logxor (bits ina-add 47 0)
               (bits inb-add 47 0)))).

```

Iteration presents a minor complication to this scheme, but the RTL loop construct may be effectively translated into LISP recursion. For example, the iterative definition of the signal `lsa` of Fig. 3 generates the following ACL2 code:

```

(defun lsa-aux (lsa found lop i)
  (declare (xargs :measure (1+ i)))
  (if (and (integerp i) (>= i 0))
      (if (not (equal (logand (bitn lop i)
                              (comp1 found 1))
                      0))
          (lsa-aux (- 68 i) found lop (1- i))
          (lsa-aux lsa 1 lop (1- i)))
      lsa))

(defun lsa (lop) (lsa-aux 0 0 lop 68))

```

Note that a *measure* declaration was inserted by hand as a hint to the prover in establishing the admissibility of the recursive definition of `lsa-aux`, but this was the only modification required of the automatically generated code.

Finally, an ACL2 function corresponding to each output of the circuit is generated, with the circuit inputs as arguments. This function is defined by means of the `let*` operator, calling in succession the functions corresponding to the circuit's wires and binding their values to variables that represent the corresponding signals. Finally, the binding of the selected output signal is returned. The function corresponding to the sole output of our simplified adder takes the following form:

```

(defun adder (a b op rc pc)
  (let* ((mana (mana a))
         (manb (manb b))
         (expa (expa a))
         (expb (expb b))
         (signa (signa a))

```



```

(signb (signb b))
.....
(r (r class-reg sign-reg exp-reg man-reg)))
r))

```

The number of bindings in this definition (i.e., the number of wires in the circuit) is 209. The translation of the actual adder RTL is similar, but much longer, involving over 700 bindings. However, the proof of equivalence of these two functions was fairly straightforward (using the ACL2 prover), and we were then able to restrict our attention to the simpler circuit without compromising our objective of establishing the correctness of the actual RTL.

While there are a number of other feasible translation schemes, the one described above was selected because (a) the correspondence between the RTL and the resulting ACL2 code is easily recognizable, and (b) the ACL2 code may be executed (and thus tested) fairly efficiently. The disadvantage of this scheme, however, is that it produces functions that are not amenable to direct formal analysis. For this purpose, some reformulation of these functions is required.

Our goal is to generate a mechanical proof of Theorem 1 by formalizing the reasoning of Section 1. Thus, we would like to be able to state and prove a lemma pertaining to a given signal, invoking previously proved results concerning other signals, without explicitly listing these previous results or stating the dependence on these other signals in the hypothesis of the lemma. This does not seem possible if our lemmas are to be statements about the ACL2 functions described above.

Our solution to this problem is based on two features of ACL2: *encapsulation*, which allows functions to be characterized by constraining axioms rather than complete definitions, and *functional instantiation*, which allows lemmas pertaining to constrained functions to be applied to other functions that can be shown to satisfy the same constraints.

Suppose that the hypothesis and conclusion of Theorem 1 are formally represented by the functions `input-spec` and `output-spec`, respectively, so that the theorem is encoded as the formula

```

(implies (input-spec a b op rc pc)
  (output-spec a b op rc pc (adder a b op rc pc))).

```

Through encapsulation, we introduce constant functions `a*`, `b*`, etc. corresponding to the inputs by executing the following ACL2 event:

```

(encapsulate ((a* () t) (b* () t) ...)
  (local (defun a* () ...))
  (local (defun b* () ...))
  ...
  (defthm inputs* (input-spec (a*) (b*) (op*) (rc*) (pc*)))).

```

Here, the definitions of `a*`, `b*`, etc. are irrelevant as long as they allow the proof of the formula `inputs*`. The result of this event is that the functions that it introduces are undefined, but constrained to satisfy `inputs*`.

Next, we define a second set of functions corresponding to the wires of the circuit. These functions are constants, derived from the first set of functions by replacing each occurrence of a signal with the corresponding constant. For example:

```
(defun sticksum* ()
  (if (equal (esub*) 0)
      (comp1 (logxor (bits (ina-add*) 47 0)
                     (bits (inb-add*) 47 0))
            48)
      (logxor (bits (ina-add*) 47 0)
               (bits (inb-add*) 47 0)))).
```

(In fact, the translator has been modified to generate these definitions as well.) The purpose of these functions is to facilitate formal reasoning about the signals of our circuit, allowing us to prove a lemma about the behavior of a signal by invoking previously proved lemmas about the signals on which it depends. Thus, to prove a lemma pertaining to the constant (`sticksum*`), we may expand its definition and invoke any relevant lemmas about (`ina-add*`) and (`inb-add*`). In this manner, tracing the proofs of Section 4 step by step, we arrive at the following result:

```
(defthm r*-spec
  (output-spec (a*) (b*) (op*) (rc*) (pc*) (r*))).
```

But simply by expanding definitions, we may also easily prove

```
(defthm r*-adder
  (equal (r*) (adder (a*) (b*) (op*) (rc*) (pc*)))))
```

and combining the last two lemmas, we trivially deduce

```
(defthm outputs*
  (output-spec (a*) (b*) (op*) (rc*) (pc*)
               (adder (a*) (b*) (op*) (rc*) (pc*)))))
```

Finally, our desired theorem may be derived from the constraint `inputs*` and the theorem `outputs*` by functional instantiation:

```
(defthm correctness-of-adder
  (implies (input-spec a b op rc pc)
            (output-spec a b op rc pc (adder a b op rc pc)))
  :hints (("goal" :use
            (:functional-instance outputs*
              (a* (lambda ()
                    (if (input-spec a b op rc pc) a (a*))))
              (b* (lambda ()
                    (if (input-spec a b op rc pc) b (b*))))
              ...))))))
```

In this final ACL2 event, a hint is provided to the prover: use the *functional instance* of the lemma `outputs*` that is produced by replacing each of the functions `a*`, `b*`, ... with a certain zero-argument *lambda expression*. Thus, the function `a*` is to be replaced by the lambda expression

```
(lambda () (if (input-spec a b op rc pc) a (a*))),
```

the value of which is

```
(if (input-spec a b op rc pc) a (a*)),
```

and the constant corresponding to each of the other inputs is similarly instantiated. Then, according to the principle of functional instantiation, the desired theorem may be established by proving two subgoals. The first is the implication that the statement of the theorem follows from the instantiated lemma:

```
(implies
  (output-spec a b op rc pc
    (adder (if (input-spec a b op rc pc) a (a*))
            (if (input-spec a b op rc pc) b (b*))
            ...))
  (implies (input-spec a b op rc pc)
    (output-spec a b op rc pc (adder a b op op rc pc))))
```

The second subgoal is the corresponding functional instance of the constraint `inputs*`:

```
(input-spec (if (input-spec a b op rc pc) a (a*))
  (if (input-spec a b op rc pc) b (b*))
  ...).
```

But the first subgoal is trivial, second follows from `inputs*` itself, and the theorem `correctness-of-adder` follows.

Acknowledgments

Several people have contributed to this project. The RTL-ACL2 translator was implemented by Art Flatau. Stuart Oberman designed the Athlon adder and explained it to the author. Matt Kaufmann and J Moore provided some helpful modifications of ACL2 and advice in its use.

References

1. Institute of Electrical and Electronic Engineers, "IEEE Standard for Binary Floating Point Arithmetic", Std. 754-1985, New York, NY, 1985.
2. Intel Corporation, *Pentium Family User's Manual, Volume 3: Architecture and Programming Manual*, 1994.

3. Kaufmann, M., Manolios, P., and Moore, J, *Computer-Aided Reasoning: an Approach*, Kluwer Academic Press, 2000.
4. Moore, J, Lynch, T., and Kaufmann, M., “A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5_K86 Floating Point Division Algorithm”, *IEEE Transactions on Computers*, 47:9, September, 1998.
5. Oberman, S., Hesham, A., and Flynn, M., “The SNAP Project: Design of Floating Point Arithmetic Units”, Computer Systems Lab., Stanford U., 1996.
6. Russinoff, D., “A Mechanically Checked Proof of IEEE Compliance of the AMD-K5 Floating Point Square Root Microcode”, *Formal Methods in System Design* 14 (1):75-125, January 1999. See URL <http://www.onr.com/user/russ/david/fsort.htm>.
7. Russinoff, D., “A Mechanically Checked Proof of IEEE Compliance of the AMD-K7 Floating Point Multiplication, Division, and Square Root Algorithms”. See URL <http://www.onr.com/user/russ/david/k/-div-sort.htm>.
8. Russinoff, D. and Flatau, A., “RTL Verification: A Floating-Point Multiplier”, in Kaufmann, M., Manolios, P., and Moore, J, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Press, 2000. See URL <http://www.onr.com/user/russ/david/acl2.htm>.
9. Russinoff, D., “An ACL2 Library of Floating-Point Arithmetic”, 1999. See URL <http://www.cs.utexas.edu/users/moore/publications/others/fp-README.html>.

An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps*

Roderick Bloem, Harold N. Gabow, and Fabio Somenzi

University of Colorado at Boulder

Roderick.Bloem@Colorado.EDU, hal@cs.Colorado.EDU, Fabio@Colorado.EDU

Abstract. We present a symbolic algorithm for strongly connected component decomposition. The algorithm performs $\Theta(n \log n)$ image and preimage computations in the worst case, where n is the number of nodes in the graph. This is an improvement over the previously known quadratic bound. The algorithm can be used to decide emptiness of Büchi automata with the same complexity bound, improving Emerson and Lei's quadratic bound, and emptiness of Streett automata, with a similar bound in terms of nodes. It also leads to an improved procedure for the generation of nonemptiness witnesses.

1 Introduction

Tarjan's classical algorithm [Tar79] for finding strongly connected components (SCCs) is not feasible for very large graphs. Even though its worst-case complexity is linear, it has to consider and label each node individually: it is an *explicit* algorithm. For many graphs encountered in applications such as model checking, time and space do not suffice to consider individual nodes.

By contrast, symbolic graph algorithms deal with sets of nodes, represented by their characteristic functions. Although the worst-case time complexity of symbolic algorithms is typically worse than that of corresponding explicit algorithms, they perform well as heuristics, so that many large problems can only be tackled symbolically. The first use of symbolic graph algorithms known to these authors is due to McMillan [McM83], in the context of model checking [McM85]. As set representation he used Binary Decision Diagrams [Bry80], a compact, canonical form that allows for efficient set manipulation.

The most important operations in symbolic graph algorithms are the set-theoretic operations union, intersection, and complementation and, above all, *image* and *preimage*. The latter two are collectively referred to as *steps*. This paper uses the number of steps as the main measure of efficiency.

Although the number of steps does not determine the time-complexity of a symbolic algorithm, it is an important measure of difficulty. As noted in [EmL89, EmS89, EmS90], algorithms that take a quadratic number of steps, such as the Emerson-Lei algorithm for Büchi emptiness [EmL89], often take much

* This work was supported in part by SRC contract 98-DJ-620 and NSF grant CCR-99-71195.

more time than algorithms that take a linear number of steps, such as the CTL model checking algorithm [EFS86]. It is therefore important to keep the number of steps that an algorithm takes low.

The subject of this paper is a symbolic algorithm for strongly connected component decomposition with applications to the Büchi and Streett emptiness problems.

The first symbolic algorithm for SCC decomposition is due to Touati et al. [TSS86]. It computes the transitive closure of a transition relation using *iterative squaring*. This computation is expensive [HKKR00], and the algorithm is limited to rather simple graphs. A more practical symbolic algorithm for SCC decomposition has been presented by Xie and Beerel [XB00]. When measured in terms of nodes, this algorithm takes $\Theta(n^2)$ steps in the worst case. We present an algorithm that is similar to that of Xie and Beerel but takes $\Theta(n \log n)$ steps in the worst case.

Our algorithm depends critically on *lockstep search*, the idea of performing backward and forward searches in the graph simultaneously. Lockstep search is used in [FSX] to update the connected components in an undirected graph after an edge has been deleted, and in [HKKR00] to decide Streett emptiness. Although these papers contain a complexity analysis that is similar to ours, they use lockstep search in a different fashion: to recompute (strongly) connected components from the two nodes of an edge that has been deleted. Both algorithms are explicit, and the complexity bounds that Henzinger and Telle prove have little relation to our bounds.

An algorithm for SCC decomposition can be used to decide emptiness of a Büchi automaton: the language of an automaton is nonempty if there is a nontrivial SCC that contains an accepting state. Emptiness checks for Büchi automata are in common use in symbolic LTL model checkers such as McMillan's SMV and in fair-CTL model checkers such as SMV and VIS [WZM02, WZM00, RT00]. The algorithm can easily be extended to deal with L -automata, which are used in Cospan [KMP00, HKK00].

All known symbolic algorithms for deciding Büchi emptiness, including the original Emerson-Lei algorithm [EL80, EKS89, APTK90], take $\Theta(n^2)$ steps in the worst case. It has been an open problem whether this bound can be improved [EKS89]. Our algorithm presents an improvement to $\Theta(n \log n)$.

In this paper we shall also show that the complexity of the Emerson-Lei algorithm can be bounded by $\Theta(|\mathcal{F}|dh)$, where \mathcal{F} is the set of acceptance conditions, d is the diameter of the graph and h is the length of the longest path in the SCC quotient graph. The complexity of the algorithm presented here cannot be bounded as a function of only d and h , and is hence incomparable to that of the Emerson-Lei algorithm. We shall discuss problems on which it performs worse. Our algorithm, like that of Xie and Beerel, potentially enumerates each nontrivial SCC, which may be expensive if the ratio of nontrivial SCCs to nodes is high. In that sense it is “less symbolic” than the other algorithms for Büchi emptiness.

Emerson and Lei [ELS⁺07] and Kurshan [Kur99] propose explicit algorithms for deciding emptiness of Streett automata. We combine their approach with our SCC-decomposition algorithm to yield a symbolic algorithm for Streett emptiness that takes $\Theta(n(\log n + p))$ steps in the worst case, where p is the number of non-Büchi fairness conditions. This improves the $\Theta(pn^2)$ bound of the symbolic algorithm for Streett emptiness of Kesten et al. [KPR08]. We also show that the new algorithm leads to a procedure for the generation of nonemptiness witnesses that improves over the ones of [GMS09, FSTK09, KPR08].

In this paper, we focus on the algorithm and its asymptotic complexity. An analysis of its performance on practical verification examples and a comparison with other algorithms are given in [ELS⁺07].

The flow of the paper is as follows. In Section 2 we present the basic definitions. In Section 3 we present an analysis of the Emerson-Lei algorithm. In Section 4 we describe the Lockstep algorithm, whose complexity we analyze in Section 5. In Section 6 we present a simple optimization and its analysis. In Section 7 we discuss an application of Lockstep to the generation of witnesses, and we conclude with Section 8.

2 Preliminaries

In this section we discuss the basic notions pertaining to graphs, symbolic algorithms, and Streett and Büchi automata. In the following, $\lg x$ is the base 2 logarithm of x , and $\log x$ is the logarithm of x where the base is an unspecified constant greater than 1, with the stipulation that $\log x = 1$ for $x \leq 1$.

2.1 Graphs and Strongly Connected Components

A (directed) *graph* is a pair $G = (V, E)$, where V is a finite set of nodes and $E \subseteq V \times V$ is a set of edges. The number of nodes of the graph is denoted by $n(G)$. We say that $(v_0, \dots, v_k) \in V^+$ is a *path* from v_0 to v_k of length k if for all i we have $(v_i, v_{i+1}) \in E$. A path is *nontrivial* if its length is not 0, and *simple* if all its nodes are distinct. The distance from v to w is the length of a shortest path from v to w , if one exists. The *diameter* $d(G)$ of a nonempty graph G is the largest distance between any two nodes between which the distance is defined. A *cycle* is a nontrivial path for which $v_0 = v_k$. The graph $H = (W, F)$ is an *induced subgraph* of G if $W \subseteq V$ and $F = E \cap (W \times W)$.

A *Strongly Connected Component (SCC)* of G is a maximal set $C \subseteq V$ such that for all $v, w \in C$ there is a path from v to w . An SCC C is *nontrivial* if for all $v, w \in C$ there is a nontrivial path from v to w . For $v \in V$, we define $\text{SCC}(v)$ to be the SCC that contains v . The number of SCCs (nontrivial SCCs) of G is denoted by $N(G)$ ($N'(G)$). The SCC-quotient graph of (V, E) is a graph (V', E') with $V' = \{\text{SCC}(v) \mid v \in V\}$ and $E' = \{(C, C') \mid C \neq C' \text{ and } \exists v \in C, v' \in C' : (v, v') \in E\}$.

Since the SCC graph is acyclic, it induces a partial order, and when we speak of a ‘maximal’ or ‘minimal’ SCC, we refer to this order. Specifically, a

Table 1. Glossary of symbols used in complexity analysis

d : diameter	h : height of the SCC quotient graph
N : number of SCCs	N' : number of nontrivial SCCs
n : number of nodes	p : number of non-Büchi fairness constraints

minimal (maximal) SCC has no incoming (outgoing) edges. The height of the SCC quotient graph (the length of its longest path) is denoted by $h(G)$. A subset $S \subseteq V$ is *SCC-closed* if for every SCC C , we have either $C \subseteq S$ or $C \cap S = \emptyset$. The intersection and union of two SCC-closed sets are SCC-closed, and so is the complement with respect to V of an SCC-closed set.

When using $n(G)$, $d(G)$, $h(G)$, $N(G)$, and $N'(G)$, we shall omit G if the graph is understood from the context. See Table 1 for a glossary of the parameters used in this paper.

Let H be a subgraph of G induced by a set of vertices that is SCC-closed. No parameter of Table 1 is larger in H than in G , with the exception of $d(H)$, which can be arbitrarily larger than $d(G)$. We now define a property that guarantees $d(H) \leq d(G)$.

A set $S \subseteq V$ is *path-closed* if any path of G that connects two vertices of S has all its vertices in S . In this case we also call the subgraph induced by S path-closed. Let S be path-closed and let H be its induced subgraph. We use these two simple facts.

1. $d(H) \leq d(G)$.
2. For $T \subseteq S$ both T and $S \setminus T$ are path-closed if no edge goes from T to $S \setminus T$ or no edge goes from $S \setminus T$ to T .

Note that in Property 1, an edge of G may go from T to $V \setminus S$ in the first case, and from $V \setminus S$ to T in the second case.

2.2 Symbolic Algorithms

We represent sets of nodes symbolically by their characteristic function. A symbolic algorithm can manipulate sets using union, intersection, complementation, image and preimage. It can also test a set for emptiness and pick an element from a set V using $\text{pick}(V)$. (The way in which the element is picked is immaterial to our exposition.) The image and preimage computations, also known as EY and EX, respectively, are defined as follows:

$$\begin{aligned}\text{img}_G(S) &= \{v' \in V \mid \exists v \in S : (v, v') \in E\}, \\ \text{preimg}_G(S) &= \{v' \in V \mid \exists v \in S : (v', v) \in E\}.\end{aligned}$$

We shall drop the subscript G if it is clear from the context.

The set of nodes that are reachable from a node v is denoted by $\text{EP } v$. It can be computed as the least fixpoint of the function $\lambda T. \{v\} \cup \text{img}(T)$. Similarly, the set

of nodes that can reach v is the least fixpoint of the function $\lambda T. \{v\} \cup \text{preimg}(T)$, and is denoted by $\text{EF } v$ [SS86].

In the remainder we shall refer to the computation of the image or preimage of a nonempty set as a *step* (computing the (pre-)image of an empty set requires no work). We shall measure the complexity of a symbolic algorithm as the number of steps that it takes. This choice is motivated by the larger cost normally incurred by image and preimage computations when compared to other operations performed on sets of states like union and intersection. The simplification afforded by concentrating on a few important operations compensates the occasional inaccuracy.

The algorithms we analyze in the following do not modify the transition relation of the automata on which they operate. Keeping the transition relation constant allows for a fairer comparison between different algorithms.

The characteristic functions manipulated by symbolic algorithms are in practice boolean functions that depend on variables ranging over the function domain. To represent a set of n states one then needs a set of at least $\lceil \lg n \rceil$ boolean variables. To represent the transition relation of a graph, twice as many variables are used. To write a formula that says “there is a node z such that x is related to z and z is related to y ” (as in the transitive closure computation) one needs three sets of variables. The cost of symbolic computations strongly depends on the number of variables used [BKK89], which explains why image and preimage are typically the most expensive operations. The algorithms examined in this paper all use the same number of variables.

2.3 Streett and Büchi Automata

A *Streett automaton* over an alphabet A is a tuple $\mathcal{A} = ((V, E), v_0, L, \mathcal{F})$ where (V, E) is a graph, $v_0 \in V$ is the initial node, $L : E \rightarrow A$ is the edge-labeling function, and $\mathcal{F} = \{(L_1, U_1), (L_2, U_2), \dots, (L_k, U_k)\} \subseteq 2^V \times 2^V$ is the acceptance condition (a set of pairs of sets of nodes) [Str86]. For technical reasons we also require that there is a path from v_0 to every node in V . The language of \mathcal{A} is nonempty if (V, E) contains a *fair cycle*: a cycle D such that for all pairs $(L, U) \in \mathcal{F}$, we have that $L \cap D \neq \emptyset$ implies $U \cap D \neq \emptyset$. We write $p(\mathcal{A})$, or simply p , for the number of distinct pairs $(L, U) \in \mathcal{F}$ for which $L \neq V$. We shall use this quantity (which is typically much smaller than n) in the complexity analysis.

A generalized *Büchi automaton* is a Streett automaton $((V, E), v_0, L, \mathcal{F})$ such that for all pairs $(L, U) \in \mathcal{F}$ we have $L = V$ [Buc62]. The language of a Büchi automaton is nonempty if there is a cycle that contains at least one state in U_i for every i . The emptiness condition for Büchi automata can easily be stated in terms of SCCs: the language is nonempty if the automaton contains an SCC C that is *fair*: it is nontrivial and $C \cap U_i \neq \emptyset$, for every i . Note that for Büchi automata p is 0.

The nonemptiness check for Streett automata can also be based on the identification of the SCCs of the automaton graph. However, if an SCC C intersects L_i , but not U_i , it is still possible for it to contain a fair cycle, provided such a cycle does not contain the states in L_i . This can be checked by subtracting

the states in L_i from C , and recursively analyzing the SCCs of the resulting subgraph.

3 Complexity of the Emerson-Lei Algorithm

Because of the practical importance of the Emerson-Lei algorithm for Büchi emptiness, we shall analyze its complexity in more detail. To our knowledge, the only known complexity bound is the obvious $\Theta(|\mathcal{F}|n^2)$ bound. In this section we sharpen the bound to $\Theta(|\mathcal{F}|dh)$. In Section [4](#), we shall contrast the bounds achieved by our algorithm with this bound.

Let $((V, E), v_0, L, \mathcal{F})$ be a Büchi automaton. The Emerson-Lei algorithm computes the set of states F that have a path to a fair cycle; the language of the automaton is empty iff $F = \emptyset$. The algorithm is characterized by the following μ -calculus formula. (See [\[Emerson1990\]](#) for an overview of the μ -calculus.)

$$F' = \nu Y. \bigcap_{(V, U) \in \mathcal{F}} \text{preimg}(\mu Z. Y \cap (U \cup \text{preimg}(Z))).$$

Evaluation of this formula requires two nested loops. The inner loop identifies the states from which there is a path that is contained in Y and reaches a state in U . It finds these states by starting with the states in $U \cap Y$ and extending the paths backward. Then, the predecessors of these states are computed by preimg to eliminate states that are not on cycles. This procedure is repeated once for every acceptance condition. We call the computation of the intersection of these $|\mathcal{F}|$ fixpoints a *pass*. The outer loop repeatedly performs passes until convergence is achieved.

Theorem 1. *Algorithm Emerson-Lei performs $\Theta(|\mathcal{F}|dh)$ steps in the worst case.*

Proof. We say that node s has distance δ to set S if the shortest distance from s to any node $s' \in S$ is δ .

Suppose the algorithm is called on a graph $G = (V, E)$. If we consider the subgraph induced by Y , then at iteration i of the inner loop, the iterate acquires those states that are at distance i from the set $U \cap Y$. It is easy to see that each iterate Y is path-closed—in fact we always have $\text{preimg}(Y) \subseteq Y$. Hence $d(Y) \leq d$. This limits the number of steps taken in the inner loop to $\mathcal{O}(d)$, and hence the number of steps in a pass to $\mathcal{O}(|\mathcal{F}|d)$.

It remains to be shown that the outer loop performs $\mathcal{O}(h)$ iterations. It is easy to see that throughout the execution of the algorithm, the subgraph induced by the iterate Y consists of a collection of SCCs of G . Each pass deletes every unfair SCC that was maximal at the start of the pass, and the algorithm halts at the end of a pass that began with no maximal unfair SCCs. Hence, it suffices to prove that an unfair SCC C that is maximal at the start of the i th pass is the first vertex of a simple path of length at least $i - 1$ in the SCC quotient graph. We do this by induction on i , as follows.

Suppose that the unfair SCC C is maximal at the start of the $i + 1$ st pass. Then C was not maximal at the start of the i th pass. Hence it precedes an SCC

C' that was maximal at the start of the i th pass and was deleted. Clearly C' is unfair, and so by induction C' contains the first node of a path of length at least $i - 1$. This gives the desired path of length at least i for C .

These three observations yield an $\mathcal{O}(|\mathcal{F}|dh)$ bound. We now prove that this bound is tight. We first give an example where $|\mathcal{F}|$ can be any integer greater than 1. We will then treat the case $|\mathcal{F}| = 1$. Consider a family of Büchi automata $\mathcal{A}_{k,l,m} = ((V, E), v_1, L, \mathcal{F})$ where $m > 1$ and

$$\begin{aligned} V &= \{v_i \mid 1 \leq i \leq k+l\} \cup \{v'_i \mid k < i \leq k+l\}, \\ E &= \{(v_i, v_{i+1}) \mid 1 \leq i \leq k\} \cup \{(v_i, v_j) \mid k < i < j \leq k+l\} \cup \\ &\quad \{(v_i, v'_i), (v'_i, v_i) \mid k < i \leq k+l\}, \text{ and} \\ \mathcal{F} &= \{(V, \{v_i \mid k < i \leq k+l, i - k - 1 \bmod m \neq q\}) \mid 0 \leq q < m\}. \end{aligned}$$

The labeling function L is immaterial. We have $d(V, E) = k + 1$, $h(V, E) = k + l$, and $|\mathcal{F}| = m$. See Fig. [1](#) for an example. On these graphs, the inner loop always performs $\Theta(k)$ steps (assuming $k > m$). Every pass executes the inner loop $|\mathcal{F}|$ times and removes only the rightmost SCC, for a total of $\Theta(l)$ passes. This gives a total of $\Theta(|\mathcal{F}|kl) = \Theta(|\mathcal{F}|dh)$ steps.

For the case of $|\mathcal{F}| = 1$ we use the graph above with two modifications: We omit the v'_i vertices and the acceptance condition is $\{(V, \{v_i \mid k < i \leq k+l\})\}$. Again, every pass removes only the rightmost SCC (which is now trivial), and we arrive at the same $\Theta(|\mathcal{F}|kl) = \Theta(|\mathcal{F}|dh)$ bound. \square

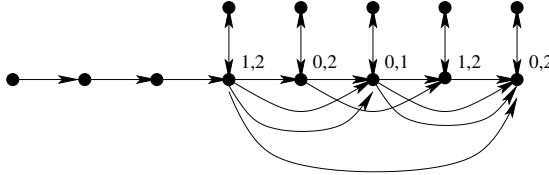


Fig. 1. The automaton $\mathcal{A}_{3,5,3}$. The states are labeled with the acceptance conditions to which they belong

Since $h \leq N$, Theorem [1](#) implies that the Emerson-Lei algorithm is also $\mathcal{O}(|\mathcal{F}|dN)$. Note that the examples from Theorem [1](#) also show that the Emerson-Lei algorithm needs $\Theta(|\mathcal{F}|n^2)$ steps.

4 Algorithm

The algorithm of Xie and Beereel [\[Xie00\]](#) picks a node v and then computes the set of nodes that have a path to v ($\text{EF } v$) and the set of nodes that have a path from v ($\text{EP } v$). The intersection of these two sets is the SCC containing v .

Both sets are SCC-closed and, after removal of the SCC, the algorithm proceeds recursively. We combine this idea with lockstep search into an algorithm that performs backward and forward search simultaneously, until one converges. This gives an $n \log n$ bound.

Fig. 1 shows the Lockstep algorithm. We shall first discuss the SCC-decomposition algorithm and then its application to Büchi and Streett emptiness. The difference between the various applications is isolated in the function Report, shown in Fig. 1.

The algorithm takes a Streett automaton as argument. If we set the acceptance condition to \emptyset then all SCCs are fair, Report never recurs, and the algorithm performs SCC decomposition.

In Line 11, we pick a node v from V . We shall compute $\text{SCC}(v)$. Sets F and B contain subsets of $\text{EP}(v)$ and $\text{EF}(v)$, respectively. In Lines 16–21, we successively approximate these sets, using the fixpoint characterizations given in Section 1 until either $F = \text{EP}(v)$ or $B = \text{EF}(v)$. Let us assume, for example, that $F = \text{EP}(v)$, but $B \neq \text{EF}(v)$. In Line 22, the intersection of B and F is a subset of the SCC containing v . The inclusion may be strict, since the largest distance from v to any node in the SCC may be smaller than the largest distance from any node in the SCC to v . (See Fig. 1.) We refine the approximation of B in Lines 28–33, in which Ffront remains empty. When $\text{Bfront} \cap F = \emptyset$, B contains all nodes in $\text{SCC}(v)$, and the intersection of F and B is the SCC containing v . We report $C = F \cap B$, and then recur. For the recursion, we split the state space in three: $B \setminus C$, $V \setminus B$, and C , and recur on the first two sets.

It is easily seen that B is SCC-closed, and hence $B \setminus C$ and $V \setminus B$ are SCC-closed, too. Correctness follows by induction. If $V = \emptyset$ then it does not contain a fair SCC. If $V \neq \emptyset$, it contains a fair SCC if and only if that SCC is either C , or included in B or $V \setminus B$.

For Büchi emptiness, Report enumerates all fair SCCs, but it never recurs. We can obviously stop exploration when a fair SCC is identified. When checking Streett emptiness, if Report finds a nontrivial SCC that does not immediately yield a fair cycle, it removes the ‘bad nodes’ and calls Lockstep on the rest of the SCC. (Cf. [15], [16].) Since Report recurs, it may report some subsets of SCCs as SCCs.

The restriction of the transition relation in Fig. 1 and Fig. 1 is purely for convenience. In practice we intersect the result of images and preimages with the argument V , so that the transition relation does not change during the computation.

5 Complexity Analysis

In this Section we shall analyze the complexity of the algorithm. We state our main result in Theorem 1 and we refine this bound for the case of SCC-decomposition and Büchi emptiness in Corollary 1.

```

1  algorithm Lockstep
2  in: Streett automaton  $\mathcal{A} = ((V, E), v_0, L, \mathcal{F})$ 
3
4  begin
5      node  $v$ ;
6      set of nodes  $F, B$ ;
7      set of nodes Ffront, Bfront,  $C$ , Converged;
8
9      if  $V = \emptyset$  then return;
10
11      $v = \text{pick}(V)$ ;
12
13      $F := \{v\}$ ; Ffront  $:= \{v\}$ ;
14      $B := \{v\}$ ; Bfront  $:= \{v\}$ ;
15
16     while Ffront  $\neq \emptyset$  and Bfront  $\neq \emptyset$  do
17         Ffront  $:= \text{img}(\text{Ffront}) \setminus F$ ;
18         Bfront  $:= \text{preimg}(\text{Bfront}) \setminus B$ ;
19          $F := F \cup \text{Ffront}$ ;
20          $B := B \cup \text{Bfront}$ 
21     od;
22
23     if Ffront  $= \emptyset$  then
24         Converged  $:= F$ 
25     else
26         Converged  $:= B$ ;
27
28     while Ffront  $\cap B \neq \emptyset$  or Bfront  $\cap F \neq \emptyset$  do
29         Ffront  $:= \text{img}(\text{Ffront}) \setminus F$ ; // One of these two fronts is empty
30         Bfront  $:= \text{preimg}(\text{Bfront}) \setminus B$ ;
31          $F := F \cup \text{Ffront}$ ;
32          $B := B \cup \text{Bfront}$ 
33     od;
34
35      $C := F \cap B$ ;
36
37     Report( $\mathcal{A}, C$ );
38
39     Lockstep( $((\text{Converged} \setminus C, E \cap (\text{Converged} \setminus C)^2), v_0, L, \mathcal{F})$ );
40     Lockstep( $((V \setminus \text{Converged}, E \cap (V \setminus \text{Converged})^2), v_0, L, \mathcal{F})$ );
41 end

```

Fig. 2. The Lockstep algorithm

```

1  algorithm Report
2  in: Streett automaton  $\mathcal{A} = ((V, E), v_0, L, \mathcal{F})$ 
3      SCC  $C \subseteq V$ 
4  begin
5      set of nodes  $C'$ ;
6
7      print SCC  $C$  identified;
8
9      if  $C$  is trivial then return;
10
11     if  $\forall i : C \cap L_i \neq \emptyset$  implies  $C \cap U_i \neq \emptyset$  then do
12         print "SCC is fair; language is not empty";
13         return
14     od;
15
16      $C' = C$ ;
17
18     for each  $i$  such that  $C \cap L_i \neq \emptyset$  and  $C \cap U_i = \emptyset$  do
19          $C' := C' \setminus L_i$ 
20     od
21
22     if  $C' \neq \emptyset$  then
23         Lockstep( $(C', E \cap (C' \times C')), v_0, L, \mathcal{F}$ )
24 end

```

Fig. 3. The Report function

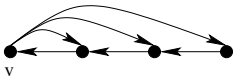


Fig. 4. two steps are needed to compute $\text{EP}(v)$, but four are needed to compute $\text{EF}(v)$. (One step in each direction to establish that a fixpoint has been reached)

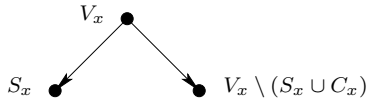


Fig. 5. A vertex x and its children in the recursion tree. Each vertex is labelled with the set V from its argument. We have $|S_x| \leq |V_x|/2$

Theorem 2. *Algorithm Lockstep runs in $\mathcal{O}(n(\log n + p))$ steps.*

Proof. We shall first analyze the complexity of the algorithm when it computes an SCC decomposition or performs a Büchi emptiness check. Then we shall extend the result to Streett emptiness, which is slightly harder because Report may call Lockstep.

We shall charge every step of the algorithm to a node, and for SCC decomposition we shall prove that we never charge more than $2 \lg n + 3$ steps to every node, bounding the number of steps by $2n \lg n + 3n$.

An invocation of Lockstep partitions V into Converged $\setminus C$, C , and $V \setminus$ Converged. One of the two sets Converged $\setminus C$ and $V \setminus$ Converged contains at most $n/2$ nodes. We refer to this set as S for small. If $|S \cup C| = k$, the number of steps in the first while loop is at most $2k$. If $S = \text{Converged} \setminus C$ this follows easily from the fact that every step acquires at least one state. If $S = V \setminus \text{Converged}$, then the non-converged set is a subset of $S \cup C$. Hence, it has fewer than $|S \cup C| = k$ states, and the first loop cannot have iterated more than k times. We charge 2 steps to every node in $S \cup C$ to account for the cost incurred in the first loop. The second while loop performs at most $|C|$ steps. We account for this by charging one step to every node in C .

To limit the total cost charged to any node, consider the recursion tree R . Every invocation of the algorithm corresponds to a vertex x of R . (See Fig. [1](#).) The argument V of that invocation is denoted by V_x , and the small set S and SCC C computed in that invocation are denoted by S_x and C_x . An arbitrary node $w \in V$ is passed down only one branch of the tree, until its SCC is identified. At the vertex of R at which $\text{SCC}(w)$ is identified, w is charged 3 steps. We now consider the charge at vertices at which the SCC of w has not been identified yet. If w is charged at vertex x of R then w is an element of S_x . Suppose that x' is the closest descendant of x in which w is charged again. We have $w \in S_{x'}$. Since $|S_{x'}| \leq |V_{x'}|/2$, we have $|S_{x'}| \leq |S_x|/2$. Hence, w is charged 2 steps at most $\lg n$ times plus 3 steps when its SCC is identified. This limits the total charge per node to $2 \lg n + 3$, and the total complexity to $2n \lg n + 3n$.

To generalize this result to Streett automata, consider a branch in the recursion tree. We still charge to a node w when it is a member of S_x or C_x for some vertex x of R . Again, w is a member of S_x at no more than $\lg n$ different vertices x , and is charged at most $2 \lg n$ steps. Unlike the Büchi fairness case, w can be a member of C_x at more than one vertex x , because Report may call Lockstep. Every time Report does so, it removes all nodes from at least one set L_i . Since Report recurs only for non-Büchi conditions, it can be invoked in the branch a total of $p + 1$ times. [1](#) Hence, we can limit the total cost charged to w when it is a member of C_x for some x to $3(p + 1)$. This bounds the total charge to any node by $2 \lg n + 3(p + 1)$, and the total cost by $2n \lg n + 3(p + 1)n$. For Büchi automata, for which p is 0, this bound simplifies to $2n \lg n + 3n$, in agreement with the previous analysis. \square

¹ This quantity can be limited to $\min(p + 1, n)$, but typically $p \ll n$.



Fig. 6. A linear graph of n nodes

It is not hard to see that $n \log n$ is an asymptotically tight bound for Büchi emptiness. Let $G(n)$ be the linear graph of n nodes. (See Fig. 1.) If Lockstep always picks the middle node then it recurs twice on $G(n/2)$, and it takes $\Theta(n \log n)$ steps. On this example, the algorithm of Xie and Beerel may take up to $\Theta(n^2)$ steps, depending on how it picks nodes.

Lockstep does not always outperform Emerson-Lei. Consider a family of graphs consisting of k identical subgraphs with no edges connecting the subgraphs. The number of steps for the Emerson-Lei algorithm is independent of k , whereas in the case of Lockstep, it grows linearly with k . On the other hand, Lockstep can take arbitrarily fewer steps than Emerson-Lei on a graph like that in Fig. 2.

The next corollary focuses on the case $p = 0$, that is, SCC analysis and Büchi emptiness.

Corollary 1. *If $p = 0$, Lockstep uses $\mathcal{O}(n \log(dN/n))$ steps.*

Proof. We shall first account for the work spent identifying the small sets of size $> d$, and then we shall account for the small sets of size $\leq d$ (for “small” as defined in the proof of Theorem 2). The rest of the work is $\mathcal{O}(n)$ since, as in the proof of Theorem 2, it is charged to the SCCs C .

Observe that when $p = 0$, each argument to Lockstep is a path-closed subgraph (by Property 1 of Section 2.1). Hence d is an upper bound on the diameter of the argument in each invocation of Lockstep. It is easy to see that this implies that the number of steps in one invocation of Lockstep is $\mathcal{O}(d)$.

We shall show that $\mathcal{O}(n)$ steps are spent processing the small sets of size $> d$. The work on a small set S is $\mathcal{O}(d)$ steps. We can account for this work by charging $\mathcal{O}(d/|S|)$ units to each node of S . It suffices to show that this results in $\mathcal{O}(1)$ units to each fixed node v . For $i \geq 0$, v belongs to at most one small set S_i whose size is between $2^i d$ and $2^{i+1} d$. The charge to v for processing S_i is $\mathcal{O}(1/2^i)$. Hence the total charge to v is less than $\sum_{i=0}^{\infty} \mathcal{O}(1/2^i) = \mathcal{O}(1)$.

We shall now account for the cost incurred identifying small sets of size $\leq d$. Let n_i , $i = 1, \dots, N$ denote the size of the i th SCC, ordered so that for all i we have $n_i \leq n_{i+1}$. Choose index s , $1 \leq s \leq N$, so that $n_i \leq d$ exactly when $i \leq s$. Since no SCC of size $> d$ is in a small set of size $\leq d$, we can limit our attention to the components indexed from 1 to s .

The proof of Theorem 2 implies that the number of steps spent processing the small sets of size $\leq d$ is

$$\mathcal{O}\left(\sum_{i=1}^s n_i \log(d/n_i)\right). \quad (1)$$

This holds because an SCC C is only involved in recursive calls on graphs with at least $|C|$ vertices. Hence, a node of S is charged at most $\log(d/|C|)$ times when it belongs to a small set with no more than d nodes.

The function $x \lg(d/x)$ is concave down. Hence Jensen's Inequality [1125] shows that the sum of (1) is maximized when all the n_i 's are equal. We distinguish two cases. First, if $n \leq ds$ then $\sum_{i=1}^s n_i \leq n$ implies the sum is maximized when $n_i = n/s$ for all $i \leq s$. In this case, (1) simplifies to $\mathcal{O}(n \log(ds/n)) = \mathcal{O}(n \log(dN/n))$. Second, if $n > ds$ then $\sum_{i=1}^s n_i \leq ds$ implies the maximum occurs when $n_i = d$ for all $i \leq s$. In this case (1) becomes $\mathcal{O}(n \log(d/d)) = \mathcal{O}(n)$. \square

The bound of Corollary 1, $\mathcal{O}(n \log(dN/n))$, implies the three bounds $\mathcal{O}(n \log d)$, $\mathcal{O}(n \log N)$, and $\mathcal{O}(dN)$. (To see this use the inequalities $N \leq n$, $d \leq n$ and $\lg x \leq x$ respectively.) Furthermore, the bound of Corollary 1 is stronger than these other bounds (e.g., take $d = N = \sqrt{n} \lg n$).

In [1260], the only bound claimed is $\mathcal{O}(dn)$, but the $\mathcal{O}(dN)$ bound is easily seen to hold for that algorithm, too.

It is not easy to generalize the result of Corollary 1 to Streett emptiness, because the diameter of the arguments cannot be expressed in terms of d when the algorithm removes nodes, and the SCCs that are found in different invocations of the algorithm are not disjoint.

Under the assumption that the number of Büchi acceptance conditions is bounded by a constant, the number of symbolic computations that the algorithm takes (including the set operations) can be bounded by the same asymptotic bound as the number of steps. Without this assumption, up to $|\mathcal{F}|n$ intersections may be used to check fairness, which is not within a constant factor of the number of steps.

6 Trimming

We will now describe a modification to the algorithm that allows us to further sharpen the bounds for Büchi emptiness.

In Line 10 of Lockstep, we can add the following code to *trim* the graph by removing the nodes that either have no path to a node in a nontrivial SCC or have no path from a node in a nontrivial SCC [1263]:

```

while  $V$  changes do
   $V := V \cap \text{img}(V) \cap \text{preimg}(V)$ ;
od.

```

We shall refer to the modified algorithm as *Lockstep with trimming*.

Note that the work added by this algorithm is limited by $\mathcal{O}(N) = \mathcal{O}(n)$ total, because every iteration of the while loop removes a trivial SCC from V . Hence, Theorem 1 and Corollary 1 remain valid with this modification.

The modification, however, allows us to prove two additional bounds for Büchi emptiness: $\mathcal{O}(dN' + N)$ and $\mathcal{O}(dN + h(N' + 1))$. Examples show that

neither of these bounds dominates the other. The former bound improves the previous $\mathcal{O}(dN)$ bound.

Theorem 3. *When $p = 0$, Lockstep with trimming takes $\mathcal{O}(\min(dN' + N, dN' + h(N' + 1)))$ steps.*

Proof. Observe that the trimming preserves the invariant that V is path-closed. Hence d is an upper bound on the diameter in each invocation of Lockstep.

First we shall prove the $\mathcal{O}(dN' + h(N' + 1))$ bound. At every invocation of the algorithm, we either pick a node in a trivial SCC or in a nontrivial SCC. We can only pick a node in a nontrivial SCC N' times. Suppose we pick a node v in a trivial SCC, and suppose again that $F = \text{Converged}$. Since we trimmed the graph before picking a node, v is on a ‘bridge’: it both has a path from and a path to a nontrivial SCC. The nontrivial SCC C that v can reach is by definition not minimal. When the algorithm recurs on F , set F is trimmed and C becomes minimal. Similarly, if $B = \text{Converged}$, a nontrivial SCC that was not maximal becomes maximal. An SCC can only become minimal or maximal once, and hence the total number of invocations in which we pick a node in a trivial SCC is limited to $N' + 1$. The cost of trimming the graphs is $\mathcal{O}(h)$ per invocation. The other costs per invocation are limited by $\mathcal{O}(d)$, as argued before, and hence the total number of steps is $\mathcal{O}(dN' + h(N' + 1))$.

The bound of $\mathcal{O}(dN' + N)$ steps holds because trimming can perform at most N steps total, and the complexity of the rest of the algorithm is bounded by dN' , as argued. \square

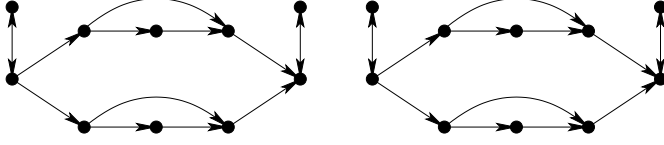
We give an example for which the $\mathcal{O}(dN' + h(N' + 1))$ bound is tight, but the $\mathcal{O}(dN' + N)$ bound is not. Consider the family of graphs G_{klm} that consist of m disjoint copies of the graph $G_{kl} = (V, E)$, defined as follows.

$$\begin{aligned} V &= \{v_0, v'_0, v_{k+1}, v'_{k+1}\} \cup \{v_{ij} \mid 0 < i \leq k, 0 < j \leq l\}, \text{ and} \\ E &= \{(v_0, v'_0), (v'_0, v_0), (v_{k+1}, v'_{k+1}), (v'_{k+1}, v_{k+1})\} \cup \{(v_0, v_{1j}) \mid 0 < j \leq l\} \cup \\ &\quad \{(v_{kj}, v_{k+1}) \mid 0 < j \leq l\} \cup \{(v_{ij}, v_{i'j}) \mid 1 \leq i < i' \leq k, 0 < j \leq l\}. \end{aligned}$$

See Fig. [1](#) for an example. For graph G_{klm} we have $h = k + 1$, $d = k + 2$, $N = m(kl + 2)$, and $N' = 2m$. If the algorithm picks node v_0 in some copy of G_{kl} initially, Lockstep will first find the SCC consisting of v_0 and v'_0 in a constant number of steps. Then, trimming takes $\Theta(k)$ steps, and identification of the SCC consisting of v_{k+1} and v'_{k+1} again takes a constant number of steps. This process is repeated m times, once for every copy of G_{kl} . This gives a total number of steps of $\Theta(km) = \Theta(dN' + h(N' + 1))$. The $\mathcal{O}(dN' + N)$ bound is not tight on this example, because N can be arbitrarily larger than k and m . Hence, $\mathcal{O}(dN' + N)$ does not dominate $\mathcal{O}(dN' + h(N' + 1))$. It is also possible to show that $\mathcal{O}(dN' + N)$ is tight and does not dominate $\mathcal{O}(dN' + h(N' + 1))$.

7 Finding Fair Paths with Lockstep

The Emerson-Lei algorithm computes a set of states that contains all the fair SCCs. If the set is not empty, it is desirable to extract a nonemptiness witness,

Fig. 7. Graph $G_{3,2,2}$

that is a path starting at the initial state and terminating in a fair cycle. The commonly used algorithm for this task is described in [Emerson-Lei], and can be shown to have complexity $\mathcal{O}(|\mathcal{F}|dh)$.

We present an algorithm that finds a maximal SCC in $\mathcal{O}(\min(dh, n))$ steps. It can be used to help find a fair cycle when using the Emerson-Lei algorithm, since Emerson-Lei constructs a subgraph where every maximal SCC is fair.

The algorithm is a variant of Lockstep called MaxSCC. It is called with arguments $((V, E), v)$, where (V, E) is a graph and $v \in V$. The algorithm begins with lines 4–10 and 12–35 of Lockstep, which identify sets F , B , C and Converged. (We omit line 11 which defines v , since v is already defined.) The rest of the algorithm is as follows:

```

if  $C = F$  then
  return  $C$ 
else
   $v := \text{pick}(\text{img}(C) \setminus C)$ ;
  if  $F = \text{Converged}$  then
     $\text{Closed} := F \setminus C$ 
  else
     $\text{Closed} := V \setminus B \setminus C$ ;
     $\text{MaxSCC}(\text{Closed}, E \cap \text{Closed}^2, v)$ 

```

The algorithm sets Closed to an SCC-closed set that contains a maximal SCC and v to a vertex of Closed , and then recurs on Closed . Since Closed is always a proper subset of (the current set) V , the algorithm eventually returns a maximal SCC. Hence the algorithm is correct.

Theorem 4. *Algorithm MaxSCC runs in $\mathcal{O}(\min(dh, n))$ steps.*

Proof. To show the bound $\mathcal{O}(dh)$ recall that lines 1–35 of Lockstep use $\mathcal{O}(d)$ steps. Furthermore the choice of v ensures that $\text{SCC}(v)$ is a successor of the SCC found in the previous invocation of the algorithm (if any). Thus there are at most h invocations, and the algorithm uses $\mathcal{O}(dh)$ steps.

To show the $\mathcal{O}(n)$ bound, Lockstep guarantees that the number of steps in one invocation is $\mathcal{O}(|B \cup C|)$. In both cases of the definition of Closed , Closed is disjoint from $B \cup C$. We charge each vertex of $B \cup C$ $\mathcal{O}(1)$ steps. This accounts for all the steps that were executed, and gives total charge $\mathcal{O}(n)$. \square

A simple variant of the algorithm consists of returning the first fair SCC found. This is normally desirable when constructing witnesses, because shorter paths are easier to examine.

Given an $\mathcal{O}(dh)$ algorithm to locate a fair SCC in the set returned by the Emerson-Lei algorithm, one can build a fair path in $\mathcal{O}(dh + |\mathcal{F}|d)$ steps. This improves on the $\mathcal{O}(|\mathcal{F}|dh)$ bound of the algorithm in [Emerson94].

A minor modification of the algorithm described in this section can find a minimal SCC with the same complexity. This version can be used with algorithms like the ones of [Emerson94, Emerson95], which return a set of states in which all minimal SCCs are guaranteed fair. In such a case, the algorithm of [Emerson94] cannot be applied directly; instead, approaches similar to that of [Emerson95] are used to isolate a fair SCC. Using Lockstep instead of those approaches improves the complexity of this phase of the computation from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

8 Conclusions

We have presented Lockstep, a symbolic algorithm for SCC decomposition with applications in deciding Büchi and Streett emptiness. We analyzed its complexity, which is characterized in terms of the number of image and preimage computations.

For SCC-decomposition and Büchi emptiness, the algorithm has a complexity of $\Theta(n \log n)$ number of steps, where n is the number of nodes in the graph. This improves the known quadratic bounds in number of nodes for both problems [Emerson94, Emerson95], although Lockstep does not consistently outperform the Emerson-Lei algorithm.

We also showed a sharper bound of $\mathcal{O}(n \log(dN/n))$ steps, which implies bounds $\mathcal{O}(n \log d)$, $\mathcal{O}(n \log N)$, and $\mathcal{O}(dN)$; a simple optimization also achieves a bound of $\mathcal{O}(\min(dN' + N, dN' + h(N' + 1)))$ steps.

For Streett emptiness, we have bounded the complexity to $\mathcal{O}(n(\log n + p))$. This improves the quadratic bound that was formerly known [Emerson94]. Finally, we have shown that Lockstep can be used to produce a nonemptiness witness in $\mathcal{O}(\min(dh, n) + |\mathcal{F}|d)$ steps.

The Lockstep algorithm uses two sets of variables in the characteristic functions (one set to encode the sources of the transitions and the other to encode the destinations) and does not require the modification of the transition relation. Both assumptions are important for runtime efficiency. Relaxing either one may lead to a reduction in the number of steps, but may render the algorithm less practical. For instance, using three sets of variables instead of two allows one to compute the transitive closure of the graph, from which the SCC of the graph can be easily derived [Emerson94]. The number of major steps is $\mathcal{O}(d)$ (even without iterative squaring), but the resulting algorithm is impractical.

The computation of the transitive closure can be seen as a maximally parallelized version of an SCC enumeration procedure like the one described in this paper. Lesser degrees of parallelization can also be achieved by either adding fewer than one full set of variables, or by pruning the transition relation so as

to conduct multiple searches in common. It remains to be ascertained whether such parallel searches lead to practical algorithms.

In their proposed scheme for complexity classes of symbolic algorithms, Hojati et al. [HHK96] use the number of sets of variables as one parameter. They do not consider arbitrary numbers of variables, and they do not consider modifications to the transition relations. The SCC decomposition algorithms that conduct a limited number of parallel searches may lead to a refinement of classification of [HHK96], or to a different approach to the classification of symbolic algorithms.

Acknowledgment

The authors thank Kavita Ravi for drawing their attention to lockstep search.

References

- [B⁺96] R. K. Brayton et al. VIS. In *Formal Methods in Computer Aided Design*, pages 248–256. Springer-Verlag, Berlin, November 1996. LNCS 1166.
- [BRS99] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.
- [CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
- [EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.
- [EL87] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.
- [ES81] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the Association for Computing Machinery*, 28(1):1–4, January 1981.
- [HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Eighth Conference on Computer Aided Verification (CAV '96)*, pages 423–427. Springer-Verlag, 1996. LNCS 1102.

- [HKS_V97] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 268–278. Springer-Verlag, Berlin, 1997. LNCS 1254.
- [HLP52] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities*. Cambridge University Press, 1952.
- [HSBK93] R. Hojati, T. R. Shiple, R. Brayton, and R. Kurshan. A unified approach to language containment and fair CTL model checking. In *Proceedings of the Design Automation Conference*, pages 475–481, June 1993.
- [HT96] M. R. Henzinger and J. A. Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In R. Karlsson and A. Lingas, editors, *Algorithm Theory: SWAT'96*, pages 16–27. Springer-Verlag, Berlin, 1996. LNCS 1097.
- [HTKB92] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω -regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.
- [KPR98] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98)*, pages 1–16, Berlin, 1998. Springer. LNCS 1443.
- [Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.
- [McM87] K. McMillan. Class project on BDD-based verification. Private Communication, E. M. Clarke, 1987.
- [McM94] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
- [McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 219–233, Berlin, September 1999. Springer-Verlag. LNCS 1703.
- [RBS] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In these proceedings.
- [RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*. Springer, Berlin, 1997.
- [Str82] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54:121–141, 1982.
- [Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.
- [TBK95] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω -automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.
- [Tho97] W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa [RS97], chapter 7, Vol. 3, pages 389–455.
- [XB99] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of the International Conference on Computer-Aided Design*, pages 37–40, San Jose, CA, November 1999.

Automated Refinement Checking for Asynchronous Processes

Rajeev Alur, Radu Grosu, and Bow-Yaw Wang

Department of Computer and Information Science, University of Pennsylvania
200 South 33rd Street, Philadelphia, PA 19104

{alur,grosu,bywang}@cis.upenn.edu

<http://www.cis.upenn.edu/~{alur,grosu,bywang}>

Abstract. We consider the problem of refinement checking for asynchronous processes where refinement corresponds to stutter-closed language inclusion. Since an efficient algorithmic solution to the refinement check demands the construction of a *witness* that defines the private specification variables in terms of the implementation variables, we first propose a construction to extract a synchronous witness from the specification. This automatically reduces individual refinement checks to reachability analysis. Second, to alleviate the state-explosion problem during search, we propose a reduction scheme that exploits the visibility information about transitions in a recursive manner based on the architectural hierarchy. Third, we establish compositional and assume-guarantee proof rules for decomposing the refinement check into subproblems. All these techniques work in synergy to reduce the computational requirements of refinement checking. We have integrated the proposed methodology based on an enumerative search in the model checker MOCHA. We illustrate our approach on sample benchmarks.

1 Introduction

Refinement checking is the problem of analyzing a detailed design, called the *implementation*, with respect to an abstract design, called the *specification*. Refinement checking is useful for detecting logical errors in designs in a systematic manner, and offers the potential of developing systems formally by adding details in a stepwise manner. Our notion of refinement is based on language inclusion, and due to its high computational complexity, traditionally refinement proofs are done manually or using theorem provers. A recent promising approach to *automated* refinement checking combines assume-guarantee reasoning with BDD-based symbolic search routines [1, 2, 3], and has been successfully applied to synchronous hardware designs such as pipelined processors [4] and a VGI chip [5]. In this paper, we develop the foundations and heuristics for automated refinement checking of *asynchronous* processes, and report on several case studies.

¹ Note that stronger notions such as bisimulation and simulation are used in many process algebras, and are supported by automated tools [6, 7, 8].

The standard notion of refinement is inclusion of trace languages: every observable behavior of the implementation must also be a possible behavior of the specification. In the asynchronous setting such as shared memory multi-processors, asynchronous circuits, and distributed processes communicating by messages, there is no notion of global time, and the speeds of the processes are independent. Consequently, behaviors that differ from each other only modulo stuttering (repetition of the same observation) need to be considered identical, and the appropriate notion of refinement is inclusion of *stutter-closed* languages: $P < Q$ iff every trace of P is stutter-equivalent to some trace of Q [1, 10].

Given two processes P and Q , checking whether $P < Q$ holds, is computationally hard: if Q is nondeterministic then it must be determinized, which requires subset construction, and furthermore, to account for stuttering, an ε -closure construction must be applied, which requires computation of the transitive closure of the transition relation. Both these problems disappear if the specification process Q has no private state. Then, as in the synchronous case, it suffices to establish that every reachable transition of P has a corresponding transition in Q , and this can be done by an on-the-fly search routine that can report counter-examples. On the other hand, when the specification has private variables, the classical approach is to require the user to provide a definition of the private variables of the specification in terms of the implementation variables (this basic idea is needed even for manual proofs, and comes in various formalizations such as refinement maps [11], homomorphisms [12], forward-simulation maps [13], and witness modules [14, 15]). Thus, the refinement check $P < Q$ reduces to $P \parallel W < Q$, where W is the user-supplied witness for private variables of Q . In our setting of asynchronous processes, it turns out that the witness W itself should not be asynchronous (that is, for asynchronous W , $P \parallel W < Q$ typically does not hold). This implies that the standard trick of choosing Q itself as a witness, used in many of the case studies reported in [16, 17], does not work in the asynchronous setting. As a heuristic for choosing W automatically, we propose a construction that transforms Q to $Eager(Q)$, which is like Q , but takes a stuttering step only when all other choices are disabled. This construction is syntactically simple, and as our case studies demonstrate, turns out to be an effective way of automating witness construction. The complexity of the resulting check, is then, proportional to the product of P and Q .

The second component of the proposed method is a heuristic for on-the-fly search based on compressing unobservable transitions in a hierarchical manner. This is an extension of our earlier work on efficient invariant verification [18]. The basic idea is to describe the implementation P in a hierarchical manner so that P is a tree whose leaves are atomic processes, and internal nodes compose their children and hide as many variables as possible. The basic reduction strategy, proposed by many researchers, is simple: while computing the successors of a state of a process, apply the transition relation repeatedly until a shared variable is accessed. This is applicable since changes to private state are treated as stuttering steps. The novelty is in applying the reduction in a recursive manner exploiting the hierarchical structure. Our strategy is easy to implement,

and gives significant reductions in space and time requirements, particularly for well-structured systems such as rings and trees.

The last component of our methodology is an *assume guarantee* principle for the stutter-closed refinement in the context of our modeling language of *reactive modules* [10]. Our assume guarantee principle asserts that to prove $P_1 \parallel P_2 < Q_1 \parallel Q_2$, it suffices to establish separately $P_1 \parallel Q_2 < Q_1$ and $Q_1 \parallel P_2 < Q_2$. This principle, similar in spirit to many previous proposals [1, 2, 3, 4, 5], reduces the verification of a composition of implementation components to individual components, but verifies an individual component only in the context of the specifications of the other components. Our first two techniques are used to check individual components afterwards.

We have incorporated our methodology using an enumerative search engine in the new implementation of the model checker MOCHA (see [11] for a description of the first release). This version is implemented in Java, and supports an extensive GUI with a proof assistant that allows the user to select refinement goals, and generates subgoals via compositional and assume-guarantee rules. The counter-examples generated by the refinement checker are displayed by a simulator in the message-sequence-chart like format. The tool is available at <http://www.cis.upenn.edu/~mocha>

The case studies reported in this paper include (1) an assume-guarantee style proof relating two descriptions of alternating-bit protocol, (2) a refinement check of a tree-structured implementation of an n -way arbiter using 2-way elements (this illustrates the use of eager witnesses, and efficiency of the heuristic for hierarchical reduction), (3) a leader election protocol (illustrating the efficiency of the hierarchical reduction), and (4) ring of distributed mutual exclusion cells [20] (illustrating use of automatic witness construction, assume-guarantee proofs, and hierarchical reduction). While these examples have been analyzed previously by model checkers, prior studies have focused on verifying temporal logic requirements, and in contrast, our tool checks (stutter-closed) refinement with respect to an abstract specification process.

2 Process Model

We start with the definition of processes. The model is a special class of *reactive modules* [10] that corresponds to asynchronous processes communicating via read-shared variables. A process is defined by the set of its variables, rules for initializing the variables, and rules for updating the variables. The variables of a process P are partitioned into three classes: *private* variables that cannot be read or written by other processes, *interface* variables that are written only by P , but can be read by other processes, and *external* variables that can only be read by P , and are written by other processes. Thus, interface and external variables are used for communication, and are called *observable* variables. The process controls its private and interface variables, and the environment controls the external variables. The separation between controlled and external variables is essential for the assume guarantee reasoning, and the separation between pri-

vate and observable variables is essential for compressing internal transitions effectively. Once the variables are defined, the state space of the process is determined. A state is also partitioned into different components as the variables are, for instance, controlled state and external state. The initialization specifies initial controlled states, and the transition relation specifies how to change the controlled state as a function of the current state.

Definition 1. A process P is a tuple (X, I, T) where

- $X = (X_p, X_i, X_e)$ is the (typed) variable declaration. X_p , X_i , X_e represent the sets of private variables, interface variables and external variables respectively. Define the controlled variables $X_c = X_p \cup X_i$ and the observable variables $X_o = X_i \cup X_e$;
- Given a set X of typed variables, a state over X is a function mapping variables to their values. Define Q_c to be the set of controlled states over X_c and Q_e the set of external states over X_e . $Q = Q_c \times Q_e$ is the set of states. We also define Q_o to be the set of observable states over X_o ;
- $I \subseteq Q_c$ is the set of initial states;
- $T \subseteq Q_c \times Q_e \times Q_c$ is the transition relation with the property (called asynchronous property) that for any $q \in Q_c$ and any $e \in Q_e$, $(q, e, q) \in T$.

■

Starting from a state q , a successor state is obtained by independently letting the process update its controlled state and the environment update the external state. The asynchronous property says that a process may idle at any step, and thus, the speeds of the process and its environment are independent.

Definition 2. Let $P = ((X_p, X_i, X_e), I, T)$ be a process, and q, q' be states. Then q' is a successor of q , written $q \xrightarrow{P} q'$, if $(q[X_c], q[X_e], q'[X_c]) \in T$.

■

Note that our model allows simultaneous updates by component processes, and thus, is different from the interleaving model (as used in SPIN [14], for instance). It is a special case of the model used by MOCHA, which supports a more general synchronous model in which the updates by the process and its environment can be mutually dependent in an acyclic manner. Modeling asynchrony within a synchronous model by a nondeterministic choice to stutter is a well-known concept [15].

In order to support structured descriptions, we would like to build complex processes from simpler ones. Two constructs, **hide** H in P and $P \parallel P'$ for building new processes are defined (we also support *instantiation*, or *renaming*, but it is not needed for the technical development in this paper). The hiding operator makes interface variables inaccessible to other processes, and its judicious use allows more transitions to be considered internal.

² For a state q over variables X , and a subset $Y \subseteq X$, $q[Y]$ denotes the projection of q on the set Y .

Definition 3. Let $P = ((X_p, X_i, X_e), I, T)$ be a process and $H \subseteq X_i$. Define the process $\text{hide } H \text{ in } P$ to be $((X_p \cup H, X_i \setminus H, X_e), I, T)$. ■

The parallel composition operator allows to combine two processes into a single one. The composition is defined only when the controlled variables of the two processes are disjoint. This ensures that the communication is nonblocking, and is necessary for the validity of the assume guarantee reasoning.

Definition 4. Let $P = ((X_p^P, X_i^P, X_e^P), I^P, T^P)$ and $Q = ((X_p^Q, X_i^Q, X_e^Q), I^Q, T^Q)$ be processes where $X_c^P \cap X_c^Q = \emptyset$. The composition of P and Q , denoted $P \parallel Q$, is defined as follows.

- $X_p = X_p^P \cup X_p^Q$; $X_i = X_i^P \cup X_i^Q$; $X_e = (X_e^P \cup X_e^Q) \setminus X_i$;
- $I = I^P \times I^Q$;
- $T \subseteq Q_c \times Q_e \times Q_c$ where $(q, e, q') \in T$ if $(q[X_c^P], (e \cup q)[X_e^P], q'[X_c^P]) \in T^P$ and $(q[X_c^Q], (e \cup q)[X_e^Q], q'[X_c^Q]) \in T^Q$. ■

It follows from the definition that the transition relation of the composed process has the asynchronous property.

A *stuttering* step is a step in which the observation does not change. The interaction of a process with the environment is not influenced by the stuttering steps it takes. To capture this aspect, we first extend the notion of successor and take only observable (non-stuttering) moves into account. A *weak successor* can be obtained by a sequence of successors where all the steps are stuttering steps except the last step.

Definition 5. Let $P = ((X_p, X_i, X_e), I, T)$ be a process, and q, q' states. We say q' is a weak successor of q , $q \xrightarrow{P}_w q'$, if there are states $q_0 = q, q_1, \dots, q_n = q'$ such that

- for all $0 \leq i < n$. $q_i[X_o] = q_0[X_o]$; and
- for all $0 \leq i < n$. $q_i \xrightarrow{P} q_{i+1}$; and
- $q_{n-1}[X_o] \neq q_n[X_o]$.

An execution of P is a sequence $\bar{\sigma} = q_0 q_1 \dots q_n$ in Q^* , where $q_0 \in I \times Q_e$ and $q_i \xrightarrow{P}_w q_{i+1}$ for $0 \leq i < n$. ■

The *trace* of an execution is its projection to observables:

Definition 6. Let $P = ((X_p, X_i, X_e), I, T)$ be a process and $\bar{\sigma} = q_0 q_1 \dots q_n$ an execution of P . The trace $\text{tr}(\bar{\sigma})$ of $\bar{\sigma}$ is a sequence in Q_o^* , defined to be $q_0[X_o] q_1[X_o] \dots q_n[X_o]$. The language of a process P , $L(P)$, is defined to be the set of traces of all executions of P . ■

Note that the language of a process completely determines its interaction with the environment: the language $P \parallel P'$ can be constructed from the languages $L(P)$ and $L(P')$.

We say process P *weakly refines* process Q if the projection of the trace of any execution of P is a trace of some execution of Q . Intuitively, P weakly refines Q if any observable behavior of P is also an observable behavior of Q modulo stuttering.

Definition 7. Let $P = ((X_p^P, X_i^P, X_e^P), I^P, T^P)$ and $Q = ((X_p^Q, X_i^Q, X_e^Q), I^Q, T^Q)$ be two processes. Define P weakly refines Q , $P < Q$, if $X_i^Q \subseteq X_i^P$, $X_o^Q \subseteq X_o^P$, and $\{\bar{\alpha}[X_o^Q] : \bar{\alpha} \in L(P)\} \subseteq L(Q)$. We write $P \cong Q$ if both $P < Q$ and $Q < P$. ■

Note that the definition of the refinement allows the implementation to have more interface variables than the specification. The refinement relation defines a preorder over processes.

3 Refinement Checking

In the refinement verification problem, the user provides the process definitions for the implementation *Impl* and specification *Spec*, and our task is to check $Impl < Spec$. The problem is computationally hard for two reasons:

1. Checking language inclusion requires determinizing the specification *Spec*. However, determinization requires subset construction, and may cause exponential blowup, and thus, should be avoided.
2. Since the definition of weak refinement considers stutter-closed traces, we need to consider the ϵ -closure of the specification *Spec* (that is, we need to match implementation transitions by the weak successors in specification). This problem is specific to the asynchronous setting. The computation of ϵ -closure is typically done by the transitive closure construction, and this is expensive.

3.1 Asynchronous Specification without Private Variables

If the specification has no private variables, all variables appear in the implementation as well. An observable implementation state corresponds to at most one state in the specification. Hence the first problem is resolved. In addition, since each specification transition is observable, the ϵ -closure of *Spec* is *Spec* itself. The refinement check then corresponds to verifying that (1) every initial state of *Impl* has a corresponding initial state of *Spec*, and (2) every reachable transition of *Impl* has a corresponding transition in *Spec*. This can be checked using the function shown in Figure ■. Notice that it is an on-the-fly algorithm and reports a violation once detected. It is easy to modify the algorithm to return, as a counter-example, the trace of *Impl* that is not a trace of *Spec*.

3.2 Specification with Private Variables

If the specification has private variables, the correspondence between implementation states and specification states should be provided by the user in order to make the checking feasible. The user needs to provide a module that assigns suitable values to the private variables of the specification in terms of values of implementation variables. This basic idea is needed even for manual

```

funct SimpleRefinement( $s, Impl, Spec$ )  $\equiv$ 
  for  $t$  such that  $s \xrightarrow{Impl} t$  do
    if  $\neg(s[X_o^{Spec}] \xrightarrow{Spec} t[X_o^{Spec}])$  then return false
    elsif  $t \notin \text{table}$  then
      insert(table,  $t$ );
      if  $\neg \text{SimpleRefinement}(t, Impl, Spec)$  then return false
  od
return true

```

Fig. 1. Algorithm for refinement checking

proofs, and comes in various formalizations such as refinement maps [11], homomorphisms [12], forward-simulation maps [13], and witness modules [14, 15]. We formalize this using *witnesses*.

Definition 8. Let $Impl = ((X_p^I, X_i^I, X_e^I), I^I, T^I)$ and $Spec = ((X_p^S, X_i^S, X_e^S), I^S, T^S)$ be two processes such that $X_o^S \subseteq X_o^I$. Then $W = ((X_p^W, X_i^W, X_e^W), T^W)$ is called a witness for $Spec$ in $Impl$ if $X_i^W = X_p^S$, $X_e^W \subseteq X_i^I$ and $T^W \subseteq Q^W \times Q^W$. We write $q \xrightarrow{W} q'$ for $(q, q') \in T^W$. ■

Notice that the transition relation of a witness takes new values of its external variables into account. That is, witnesses are not asynchronous processes, and can proceed synchronously with the implementation.

Since our parallel composition was defined over processes, we need a new operation corresponding to product with the witness. We denote this operation by \otimes (its formal definition is omitted due to lack of space). Once a proper witness is found, the refinement can be established by the following theorem.

Theorem 1. Let $Impl, Spec = ((X_p^S, X_i^S, X_e^S), I^S, T^S)$ be processes and W be a witness for $Spec$ in $Impl$. Define $Spec^u = ((\emptyset, X_p^S \cup X_i^S, X_e^S), I^S, T^S)$. Then $Impl \otimes W < Spec^u$ implies $Impl < Spec$. ■

If the verifier has expertise on the implementation, an efficient witness can be built based on this expert knowledge. Ideally, the witness should be *stateless*, and thus, should define the values of X_p^S as a *function* of the variables X^I . However, if the implementation is complicated, finding a proper witness may be difficult. In this case, one would like heuristics to fabricate the witness automatically.

Suppose the specification is composed of two subprocesses which control private and interface variables separately, say $Spec = Spec_p \parallel Spec_o$. The one controlling private variables ($Spec_p$) would be a candidate as a witness, for it updates the private variables of $Spec$ based on the values of the observables of $Spec$, and hence of $Impl$. This is used in many of the case studies reported in [16, 17], and in fact, this choice of witness is complete if $Spec_p$ is deterministic. However, in our setting, $Spec_p$ is asynchronous, and is always nondeterministic as it may stutter at each step. In our examples, setting witness to $Spec_p$ does not work. Consider

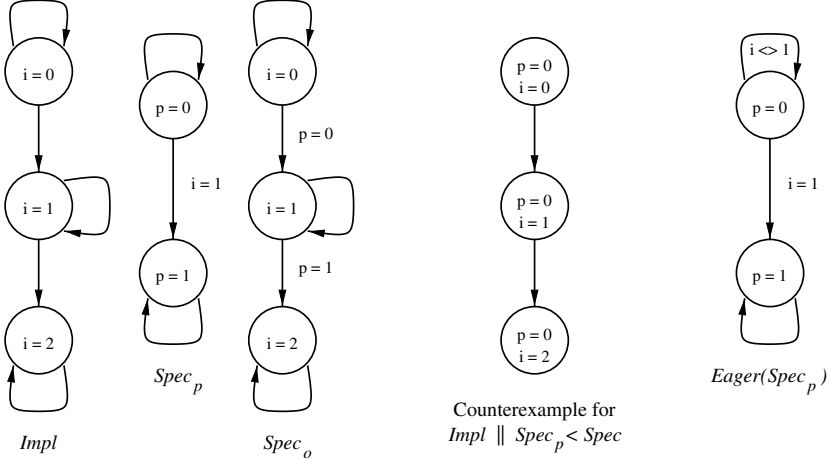


Fig. 2. Illustration of Eager Witness

the example shown in Figure 2. Three processes $Impl$, $Spec_p$ and $Spec_o$ appear in the left. Suppose the variable p is private and i is an interface variable. It is easy to see that $Impl < Spec_p \parallel Spec_o$. If we were to use $Spec_p$ as a witness (assuming variables have their proper types), the middle figure gives a trace in $Impl \parallel Spec_p$ but not in $Spec_p \parallel Spec_o$. This problem results from the asynchronous property: private variables may not be updated because it is always possible to retain their old values regardless of observable variables. To avoid this, we would like them to react to the implementation updates immediately, whenever possible. This motivates the following definition:

Definition 9. Let $P = ((X_p, X_i, X_e), I, T)$ be a process. Define the eager variant of P , $Eager(P)$, to be the process $((\emptyset, X_p \cup X_i, X_e), I, Eager(T))$ where $Eager(T) = T \setminus \{(q, e, q) : \text{there exist } q' \neq q. (q, e, q') \in T\}$. ■

In our modeling language, the eager variant of an asynchronous process can be constructed by a simple syntactic translation: we simply need to remove the keyword *lazy* from the atom declarations.

We propose to use $Eager(Spec_p)$ as a witness, and thus, the refinement check $Impl < Spec_p \parallel Spec_o$ is reduced to $Impl \otimes Eager(Spec_p) < Spec_p \parallel Spec_o$. The right figure in Figure 2 shows the synchronous witness $Eager(Spec_p)$, and it is easy to verify that $Impl \otimes Eager(Spec_p) < Spec_p \parallel Spec_o$ holds.

The assumption that $Spec$ is decomposed into $Spec_p$ and $Spec_o$ can be relaxed if one can extract the private component $Spec_p$ from any given specification process $Spec$ in a systematic way. Consider the transition shown in Figure 2 where i is an interface variable and p a private variable. We would like to construct a process $Priv(Spec)$ that controls only p . Notice that variable i becomes an external variable in $Priv(Spec)$, and cannot be updated by it. Now the private variable p should be updated whenever i changes from 0 to 1. Since this ex-

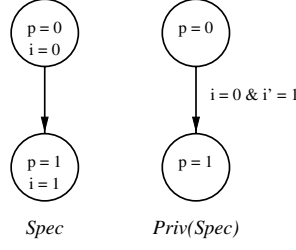


Fig. 3. Extraction of *Spec* Controlling Private Variables

traction is used only to construct an eager witness, it can be synchronous, and can read new values of i . Hence we can construct a corresponding transition in $Priv(Spec)$ as shown in the figure. This translation also can be automated easily in our modeling language by a simple syntactic transformation.

4 Hierarchic Reduction of Unobservable Transitions

Our reduction scheme is based on compressing sequences of transitions that neither read nor write observable variables. Such transitions are called *unobservable transitions* as formalized in the following definition.

Definition 10. Let $P = (X, I, T)$ be a process. A transition $(q, e, q') \in T$ is invisible if

- it doesn't write to interface variables. $q[X_i] = q'[X_i]$; and
- it doesn't read from external variables. For all $e' \in Q_e$. $(q, e', q') \in T$.

A transition is visible if it is not invisible. ■

It is worth emphasizing that invisibility is different from stuttering. It turns out that to check weak refinement, the number of invisible transitions taken by a process is not essential. Hence, we define a derivative of the process, which merges several invisible transitions together.

Definition 11. Let $P = (X, I, T)$ be a process. The process $NEXT\ P = (X, I, T')$ satisfies the following condition: $(q, e, q') \in T'$ if $q = q'$ or there are states $q_0 = q, q_1, \dots, q_n = q' \in Q_c$ such that $(q_i, e, q_{i+1}) \in T$ is invisible for $0 \leq i < n - 1$; and $(q_{n-1}, e, q_n) \in T$ is visible. ■

A useful property of the $NEXT$ operator is that the weak refinement is a congruence relation with respect to it. In particular, $NEXT\ P \cong P$. Therefore, one may apply the $NEXT$ operator to composite processes hierarchically. For example, instead of applying the $NEXT$ of the composition of two processes, we can apply $NEXT$ to the component processes and then compose them. In practice, the number of states of two $NEXT$ processes composed together is less than the

NEXT of the composition. This is due to the fact that NEXT can potentially reduce the number of interleavings of unobservable transitions of the components. Hence the number of intermediate states generated by composition is reduced, as will be illustrated in the examples of Section 6.

In [4], we had reported an on-the-fly algorithm to search such process expressions. A modification of that algorithm is used during refinement check. The algorithm is guaranteed to visit every reachable state of the given expression, and visits no more, and typically much less, than that required to explore the flat expression obtained by removing the applications of NEXT. The running time is linear in the number of states (more precisely, the transitions) visited by the algorithm, and there is basically no overhead in applying the reduction.

Since our modeling language distinguishes private, interface and external variables, our implementation can utilize this information to determine whether a transition is visible or not. In addition, the attributes of variables are syntactically determined, so the visibility of each transition can be recognized by parsing the model. Checking whether any transition is visible or not becomes a simple table lookup at runtime.

5 Compositional and Assume Guarantee Reasoning

A key property of the weak refinement relation is *compositionality*. It ensures that the refinement preorder is congruent with respect to the module operations.

Proposition 1. (*Compositionality*) *If $P < Q$ then $P \parallel R < Q \parallel R$ and $\text{hide } H \text{ in } P < \text{hide } H \text{ in } Q$.*

By applying the compositionality rule twice and using the transitivity of refinement it follows that, in order to prove that a complex compound module $P_1 \parallel P_2$ (with a large state space) implements a simpler compound module $Q_1 \parallel Q_2$ (with a small state space), it suffices to prove (1) P_1 implements Q_1 and (2) P_2 implements Q_2 . We call this the *compositional proof rule*. It is valid, because parallel composition and refinement behave like language intersection and language containment, respectively.

While the compositional proof rule decomposes the verification task of proving implementation between compound modules into subtasks, it may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2 , and vice versa. For example, consider a simple version of the *alternating bit protocol*.

Figure 1 shows the specification of the sender. The keywords *external*, *interface* and *private* classify the variables of a module as required in Definition 1. The transition relation of a module is given by several *atoms*, each atom having an exclusive update right on a subset of the controlled variables. In the asynchronous case, atoms can be viewed as (atomic) processes. The way an atom initializes and updates its controlled variables is given by a *guarded command*. The primed notation stands for new values of variables. The *reads* and *controls*


```

type Pc is {snd, wait}
type message is {a, b, c}

module senderSpec is
  external ack : channel[1] of bool
  interface abp : channel[1] of bool; msg : channel[2] of messages;
    pcS : Pc; x, y : bool
  private m : message
  lazy atom sndSp
    controls send(abp), pcS, x, y, m
    reads receive(ack), pcS, x, m
  init
    [] true -> pcS' := snd; x' := false; y' := false; m' := nondet
  update
    [] pcS = snd -> send(abp, x); send(msg, m); pcS' := wait
    [] pcS = wait & ¬isEmpty(ack) ->
      receive(ack, y); x' := ¬x; m' := nondet; pcS' := snd

```

Fig. 4. Specification of Sender Process

keywords allow to define the variables read and written by an atom. The *lazy* keyword corresponds to our asynchronous condition and allows stuttering.

To simplify the asynchronous specification style, we support a predefined *channel type*. This is essentially a record type, consisting of a ring buffer, a sender and a receiver pointer. Note however, that in our setting, the fields of this record may be controlled by different modules (or atoms) and this has to be made explicit. The sender specification simply assumes that the sender and the receiver are synchronized with each other by using the boolean one element channels **abp** and **ack**. The implementation of the sender (Figure 4) makes sure that the sender works properly in an environment that may lose messages. Each time the sender sends a new message along the channel **msg** it toggles the bit **x** and sends it along the channel **abp**. If it does not receive this bit back along the channel **ack** it resends the message. The receiver module works in a symmetric way.

Trying to prove now that **senderImp** < **senderSpec** or that **receiverImp** < **receiverSpec** fails. The implementation of the sender and receiver refine their abstract counterparts only in an environment that behaves like the abstract receiver and sender respectively. For such cases, an *assume-guarantee* proof rule is needed [24, 25, 26]. Our rule differs from the earlier ones in that it uses a different notion of refinement, namely, the stutter-closed one.

Proposition 2. (*Assume-Guarantee*) *If $P_1 \parallel Q_2 < Q_1 \parallel Q_2$ and $Q_1 \parallel P_2 < Q_1 \parallel Q_2$, then $P_1 \parallel P_2 < Q_1 \parallel Q_2$.*

Since $P_1 \parallel P_2$ has the largest state space, both proof obligations typically involve smaller state spaces than the original proof obligation. The assume-guarantee proof rule is circular; unlike the compositional proof rule, it does not

```

module senderImp is
  external ack : channel[2] of bool
  interface abp : channel[2] of bool; msg : channel[2] of message;
    pcS : Pc; x, y : bool
  private m : message
  lazy atom sndSp
    controls send(abp), pcS, x, y, m
    reads receive(ack), pcS, x, y
  init
    [] true -> pcS' := snd; x' := false; y' := false; m' := nondet
  update
    [] pcS = snd -> send(abp, x); send(msg, m); pcS' := wait
    [] pcS = wait & first(ack, ¬x) -> receive(ack, y); pcS' := snd
    [] pcS = wait & first(ack, x) ->
      receive(ack, y); x' := ¬x; m' := nondet; pcS' := snd

```

Fig. 5. Implementation of Sender Process

simply follow from the fact that parallel composition and implementation behave like language intersection and language containment. Rather the proof of the validity of the assume-guarantee proof rule proceeds by induction on the length of traces. For this, it is crucial that every trace of a module can be extended. An alternative version of the rule states that “if $P_1 \parallel Q_2 < Q_1$ and $Q_1 \parallel P_2 < Q_2$, then $P_1 \parallel P_2 < Q_1 \parallel Q_2$.” For the enumerative checker, we prefer the variant stated in Proposition 2 because it constrains the right hand side and therefore reduces the required computation.

The soundness of assume-guarantee depends crucially on the asynchronous property of the processes (it is not valid for arbitrary reactive modules). Recall that witnesses are not processes. This implies that witnesses can be introduced only after generating all subgoals using assume-guarantee since the assume guarantee requires asynchronicity.

6 Case Studies

6.1 Tree-Structured Parity Computer

We will demonstrate our proposed techniques in a tree-structured process that computes a function, say, *parity*, of the requests received from the leaf clients, where each request supplies value that is either 0 or 1 (Figure 1). Each client sends requests and receives acknowledgments to and from its parent. Each link process gathers requests from its children, computes the parity, and reports it to its parent. The root process calculates the final result and sends the result to the children link processes, which in turn, propagate the result down.

Let us focus on the process *System* described as

$$\text{Root} \parallel \text{Link} \parallel \text{Link0} \parallel \text{Link1} \parallel \text{Link00} \parallel \text{Link01} \parallel \text{Link10} \parallel \text{Link11}$$

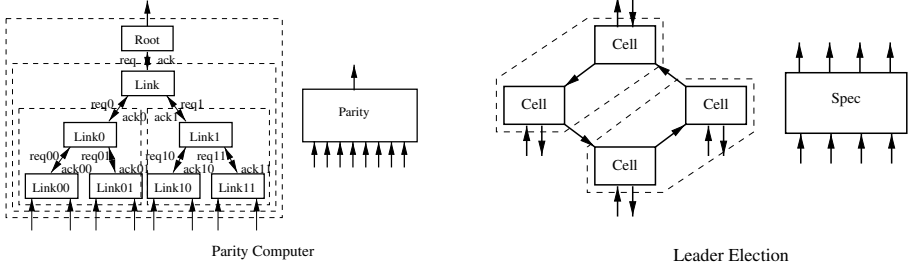


Fig. 6. Parity Computer and Leader Election

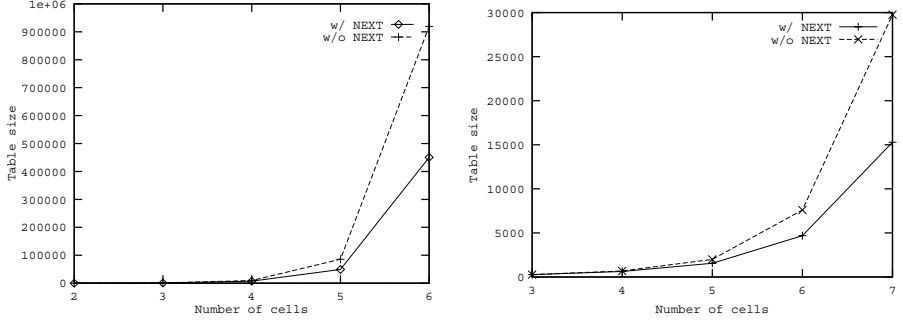


Fig. 7. Table Size of Parity Computer and Leader Election Examples

We can naturally cluster subprocesses as shown in Figure 4, replace the process *System* by the process *NextSystem*:

$$\text{NEXT hide [Root \parallel NEXT hide [Link \parallel \begin{array}{l} \text{NEXT hide (Link0 \parallel Link00 \parallel Link01)} \\ \text{NEXT hide (Link1 \parallel Link10 \parallel Link11)} \end{array}]]}.$$

For readability the argument variables to **hide** are not shown above, but should be obvious from context: only the variables communicating with the parent need to be observable at each level of the subtree. Note that a step of the tree rooted at *Link* is composed of a step of the process *Link*, and a sequence of steps of the subtree rooted at *Link0* until *req0* or *ack0* is accessed, and a sequence of steps of the subtree rooted at *Link1* until *req1* or *ack1* is accessed. Since each node communicates with parent only after it has received and processed requests from its children, we get excellent reduction. Furthermore, the congruence property ensures that *System* can be replaced by *NextSystem* during any refinement check.

The specification *Parity* is an n -way process that simply calculates the parity based on all of its inputs. However, since the input signals may not arrive at the same time, *Parity* needs private variables to record processed signals. Our goal is to check if $\text{System} < \text{Parity}$ holds. In this case, the correspondence between the private variables of the two descriptions is unclear. To construct a witness, the first step is to extract the process $\text{Priv}(\text{Parity})$ from *Parity*, and then, use its eager version as the witness. This automatic construction turns out

to be adequate in this case. Then we apply the hierarchic reduction and check $NextSystem \parallel Eager(Priv(Parity)) < Parity$. Figure 4 shows the number of states stored in the table when we check the refinement. As indicated, our hierarchical reduction is quite effective in alleviating the state-explosion problem.

6.2 Leader Election

The leader election problem consists of cells, each with a unique initial number, connected in a ring (see Figure 1). The problem is to elect a leader, which is achieved by determining who has the greatest number. Each cell can have one of three statuses: **unknown**, **chosen**, and **not_chosen**. Initially, all cells have status **unknown**. In the end, only the cell with the greatest number should have status **chosen**, and all other cells should have status **not_chosen**. Variants of this problem have been extensively used to illustrate temporal logic model checking. We show consistency of a distributed leader election protocol with respect to the process *Spec* that determines all statuses in a centralized manner by comparing the given numbers. Since there is no private variable in *Spec*, we can directly check if $System < Spec$. The result is shown in Figure 4, which indicates effectiveness of the hierarchical reduction (see Figure 4 for clustering for the hierarchical reduction).

6.3 Distributed Mutual Exclusion

In Distributed Mutual Exclusion (DME) [21], a ring of cells are connected to a user process each. Figure 5 shows the specification of a cell.

In the specification, a virtual token is passed around the cells clockwise. Initially, only one cell has the token. Each cell can send requests to its right neighbor (**right_req**) and acknowledgments to its left neighbor and user (**left_ack** and **user_ack**). When it receives a request from its left neighbor, it checks if it possesses the token. If it does not, a request is sent to its right neighbor. Otherwise, it passes the token by setting **left_ack** to true. After its left neighbor receives the token, it resets left acknowledgment to false.

It handles user's requests similarly. However, when the user releases the token by assigning **user_req** to false, the cell resets not only the user acknowledgment to false, but also the token variable to true to obtain the token from the user. Finally, if it sends request to its right neighbor and the neighbor passes the token, it sets the token variable to true and the right acknowledgment to false.

Each cell is implemented by basic logic gates (10 ANDs, 2 ORs, 7 NOTs) and special components (C-element and 2-way Mutual Exclusion blocks). The user process simply sends requests and receives acknowledgments accordingly. Previous verification studies of DME implementation have considered checking requirements written in a temporal logic such as CTL. We define the process *System* to be the composition of three cells and user processes. The specification process *Spec*, then, is the high-level distributed algorithm shown in figure 5. Our goal is to check whether $System < Spec$.

```

module CSpec.T is
  external left_req, right_ack, user_req : bool
  interface left_ack, right_req, user_ack : bool; token : bool
  lazy atom CSPEC.T
    controls left_ack, right_req, user_ack, token
    reads left_req, left_ack, right_req, right_ack,
      user_req, user_ack, token
init
  [] true -> left_ack' := false; right_req' := false;
    user_ack' := false; token' := true
update
  [] left_req & ¬left_ack & ¬right_ack & ¬token -> right_req' := true
  [] left_req & ¬left_ack & token -> left_ack' := true; token' := false
  [] left_ack & ¬left_req -> left_ack' := false
  [] user_req & ¬user_ack & ¬right_ack & ¬token -> right_req' := true
  [] user_req & ¬user_ack & token -> user_ack' := true; token' := false
  [] user_ack & ¬user_req -> user_ack' := false; token' := true
  [] right_req & right_ack -> token' := true; right_req' := false

```

Fig. 8. Specification of a DME Cell with Token

In order to make checking $System < Spec$ feasible, we would like to avoid non-deterministic choices of variable values in $Spec$. Hence, we need to establish the relation among variables of $System$ and of $Spec$. Notice that the high level specification $Spec$ uses the virtual token. Because the implementation $System$ consists of circuit signals, it is rather difficult, if not impossible, to establish the desired relation. We therefore apply the technique in section 4.1 to extract the witness module for each cell.

In figure 1, module $Wit.T$ is the witness module $Eager(Priv(Cell.T))$. It is easy to see that the witness can be generated by syntactic transformation (cf. figure 2). We thus define the witness module $Eager(Priv(Spec))$ to be the composition of cell witnesses and check $System \otimes Eager(Priv(Spec)) < Spec$. We can apply hierarchical reduction on the refinement checking. Alternatively, we may apply assume guarantee proof rule first, obtain the following three proof obligations, and then generate witnesses for each obligations.

Figure 3 shows the table size of each experiments. We can see that NEXT operator reduces the number of states while checking the whole system. There are three obligations in assume guarantee proof rule for $System$ consists of three cells. They all use the same number of states, which is less than checking the whole system. However, NEXT operator does not save space for obligations.

7 Conclusions

We have proposed a methodology, with an associated tool, for checking refinement based on stutter-closed language inclusion. Our notion of refinement is the

```

module Wit_T is
  external left_req, right_ack, user_req : bool;
           left_ack, right_req, user_ack : bool
  interface token : bool
  atom WIT_T
    controls token
    reads left_req, right_req, right_ack, user_req, user_ack, token
    awaits left_ack, user_ack, right_req
  init
    [] true -> token' := true
  update
    [] left_req & ¬left_ack & token & left_ack' -> token' := false
    [] user_req & ¬user_ack & token & user_ack' -> token' := false
    [] user_ack & ¬user_req & ¬user_ack' -> token' := true
    [] right_req & right_ack & ¬right_req' -> token' := true

```

Fig. 9. Witness Module for DME Cell

	with NEXT		without NEXT
without assume guarantee	4128		6579
with assume guarantee	$Cell_0 Spec_1 Spec_2 < Spec_0$	3088	3088
	$Spec_0 Cell_1 Spec_2 < Spec_1$	3088	
	$Spec_0 Spec_1 Cell_2 < Spec_2$	3088	

Fig. 10. Table Size of Distributed Mutual Exclusion

natural one for asynchronous processes, and in many cases, writing the specification as a process is more intuitive than writing a set of temporal logic assertions. Let us summarize our approach to refinement checking.

1. The input refinement problem concerning weak refinement of asynchronous descriptions is decomposed into subgoals using compositional and assume-guarantee rules.
2. For each subgoal of the form $Impl < Spec$ is replaced by the check $Impl \otimes W < Spec$, where W is a (synchronous) witness. The witness may either be supplied by the user, or automatically chosen to be $Eager(Priv(Spec))$.
3. Checking $Impl \otimes W < Spec$ corresponds to a reachability analysis. During the search, $Impl$ is optimized by replacing each subexpression E in $Impl$ by NEXT E .

We have reported on an enumerative refinement checker and a proof assistant. Note that the first and the third items above exploit the hierarchical constructs in our language. The methodology is illustrated based on traditional benchmarks involving asynchronous processes. We conclude by summarizing our experience about different heuristics.

Assume-guarantee reasoning. Its application requires decomposing the specification into components, and this can be tedious. The subgoals can be of much lower complexity than the original goals. However, particularly in the context of enumerative algorithms, subgoals can also be more difficult to check since specifications tend to be nondeterministic or partial. Thus, effectiveness of this technique has been mixed.

Automatic witness construction. We have proposed a simple heuristic for constructing a synchronous witness from the specification. This approach has been surprisingly effective in our examples.

Hierarchical reduction. The algorithm to compress invisible transitions in a recursive manner is easy to implement with no noticeable overhead, and good reduction in many cases. This method is incomparable to and compatible with symmetry reduction [14]. When compared to the partial-order reduction method [15], for the problem of invariant verification our method gives less reduction at least in the leader election example. However, it is unclear if partial-order reduction is applicable for refinement checking in presence of synchronous witnesses.

Acknowledgments

We thank all the members of the MOCHA team at University of California at Berkeley and at University of Pennsylvania for their assistance. This research was supported in part by NSF CAREER award CCR97-34115, by DARPA/NASA grant NAG2-1214, by SRC contract 99-TJ-688, by Bell Laboratories, Lucent Technologies, and by Alfred P. Sloan Faculty Fellowship.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. M. Abadi and L. Lamport. Composing specifications. *ACM TOPLAS*, 15(1):73–132, 1993.
3. M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17:507–534, 1995.
4. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
5. R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV’98: Computer Aided Verification*, LNCS 1427, pp. 516–520, 1998.
6. R. Alur and B.-Y. Wang. “Next” heuristic for on-the-fly model checking. In *CONCUR’99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pp. 98–113, 1999.
7. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of finite-state systems. *ACM Trans. on Programming Languages and Systems*, 15(1):36–72, 1993.
8. E. Emerson and A. Sistla. Symmetry and model checking. In *CAV’93: Computer-Aided Verification*, LNCS 697, pp. 463–478, 1993.

9. P. Godefroid. Using partial orders to improve automatic verification methods. In E. Clarke and R. Kurshan, editors, *CAV'90: Computer-Aided Verification*, LNCS 531, pp. 176–185, 1990.
10. O. Grümberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
11. T. Henzinger, X. Liu, S. Qadeer, and S. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD'99: International Conference on Computer-aided Design*, pp. 494–499, 1999.
12. T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98: Computer-aided Verification*, LNCS 1427, pp. 521–525, 1998.
13. T. Henzinger, S. Qadeer, and S. Rajamani. Assume-guarantee refinement between different time scales. In *CAV'99: Computer-aided Verification*, LNCS 1633, pp. 208–221, 1999.
14. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
15. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
16. C. Ip and D. Dill. Verifying systems with replicated components in $\text{mur}\varphi$. In *Computer Aided Verification*, LNCS 1102, 1996.
17. R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
18. N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.
19. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pp. 137–151, 1987.
20. A. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *Chapel Hill Conference on Very Large Scale Integration*, pp. 245–260, 1985.
21. K. McMillan. A compositional rule for hardware design refinement. In *Computer-Aided Verification*, LNCS 1254, pp. 24–35, 1997.
22. K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV'98: Computer-Aided Verification*, LNCS 1427, pp. 110–121, 1998.
23. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
24. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV'94: Computer Aided Verification*, LNCS 818, 1994.
25. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, LNCS 224, pp. 510–584, 1986.
26. J. Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Oxford University, 1998.
27. E. Stark. A proof technique for rely-guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pp. 369–391, 1985.

Border-Block Triangular Form and Conjunction Schedule in Image Computation^{*}

In-Ho Moon, Gary D. Hachtel, and Fabio Somenzi

Department of Electrical and Computer Engineering
University of Colorado at Boulder
{moon, hachtel, fabio}@colorado.edu

Abstract. Conjunction scheduling in image computation consists of clustering the parts of a transition relation and ordering the clusters, so that the size of the BDDs for the intermediate results of image computation stay small. We present an approach based on the analysis and permutation of the dependence matrix of the transition relation. Our algorithm computes a bordered-block lower triangular form of the matrix that heuristically minimizes the active lifetime of variables, that is, the number of conjunctions in which the variables participate. The ordering procedure guides a clustering algorithm based on the affinity of the transition relation parts. The ordering procedure is then applied again to define the cluster conjunction schedule. Our experimental results show the effectiveness of the new algorithm.

1 Introduction

Symbolic algorithms for model checking [1] spend most of the time computing the predecessors or successors of sets of states. The algorithms for these *image* and *preimage* computations have been the object of careful study over the last decade. The state transition graph of the system subjected to symbolic model checking is given as a predicate $T(y, w, x)$ that is true if there is a transition from State x to State y under Input w . The predicate is usually described by a Binary Decision Diagram [2]. Representing $T(y, w, x)$ by a single formula is often impractical [3, 4, 5]; a partitioned representation is used in those cases.

The partitioned transition relation approach is especially natural when the system to be analyzed is a deterministic hardware circuit. Then, each binary memory element of the circuit gives rise to one term of the transition relation. When the circuit is synchronous, the partitioning is conjunctive, and T can be written as the product of *bit relations*.

Efficient computation of images and preimages for partitioned transition relations requires that two problems be addressed:

1. The clustering of the bit relations so that the number of the conjuncts is reduced without negatively affecting the size of the BDDs involved.
2. The ordering of the clusters so as to minimize the intermediate products.

^{*} This work was supported in part by SRC contract 98-DJ-620 and NSF grant CCR-99-71195.

Clustering and ordering can be seen as two aspects of the general problem of arranging the bit relations in a directed tree such that its leaves are the bit relations themselves and the set of states, S , whose image or preimage must be computed. Each internal node of the tree corresponds to a conjunction and to the quantification of some variables from its result.

The subtrees from which S is not reachable can be processed once, before images or preimages are computed: They correspond to the clusters. After clustering, the internal nodes of the directed tree form a total order, which yields the desired ordering of the clusters. Therefore, we refer to the combined problem of ordering and clustering the bit relations as the *conjunction scheduling* problem.

The variables that can be quantified at each node of the tree are those that only appear in the functions at the leaves reachable from the node. It is usually advantageous to quantify many variables soon, because the ensuing reduction in the support of the conjuncts helps keep the size of the BDDs under control. Therefore, ordering and clustering the bit relations is often seen as the problem of finding a good *quantification schedule*. However, early quantification is only one objective of good ordering and clustering. Hence, we prefer the name *conjunction schedule*.

In this paper we propose an approach to the *conjunction scheduling* problem that is based on the analysis and permutation of the dependence matrix of the system. The dependence matrix gives the variables on which every bit relation depends. (These variables form the *support* of the bit relation.) Our method produces good quantification schedules, though it takes into account other factors as well, which may significantly affect the time and memory requirements of image and preimage computation. Though the main ideas presented in this paper apply to preimage computation as well as to image computation, our discussion and experiments are currently restricted to the latter.

The importance of the quantification schedule was recognized in [1, 2]. Geist and Beer [3] proposed a heuristic algorithm, later improved by Ranjan et al. [4]. Hojati et al. [5] showed that the problem of finding a tree such that the support of the largest intermediate product is less than a given constant is NP-complete under the simplifying assumption that the support of $f \wedge g$ is the union of the supports of f and g .

As a result, the algorithms that are commonly in use for clustering and ordering transition relations are heuristic. Ours is no exception. On sparse dependence matrices it runs in time linear in the size of the matrix, that is, quadratic in the number of state variables. The dependence matrices of large circuits are almost invariably sparse.

The study of the dependence matrix in image computation was the topic of [6]. The context there was image computation with the transition function method—input splitting in particular [7, 8]. From the dependence matrix one could extract information about the BDD variable order and the choice of the splitting variable. More recently, Moon et al. [9] have used the dependence matrix to dynamically guide the choice of image computation algorithm.

The rest of this paper is organized as follows. Section 1 reviews background material. Section 2 discusses the conjunction scheduling problem and presents our ordering algorithm, while Section 3 describes our clustering procedure. Section 4 discusses the benefits of decomposing the dependence matrix in connected components. Section 5 contrasts our approach to another ordering algorithm based on analysis of the dependence matrix. Section 6 presents experimental results and Section 7 concludes.

2 Preliminaries

We are interested in computing the set of successors of a set of states of a transition structure $K = \langle T, I \rangle$ consisting of a transition relation $T(y, w, x)$ and an initial predicate $I(x)$. The set of variables $x = \{x_1, \dots, x_m\}$ encode the present states; the set of variables $y = \{y_1, \dots, y_m\}$ encode the next states; and the set of variables $w = \{w_1, \dots, w_{n-m}\}$ encode the primary inputs. We assume that the transition relation is given as a product of bit relations.

$$T(y, w, x) = \bigwedge_{1 \leq i \leq m} T_i(y_i, w, x) = \bigwedge_{1 \leq i \leq m} (y_i \wedge \delta_i(w, x) \vee \neg y_i \wedge \neg \delta_i(w, x)) .$$

Each T_i is called a *bit relation*. The extension to the conjunction of arbitrary subrelations is left to the reader. Ordering and clustering rewrite the transition relation as

$$T(y, w, x) = \bigwedge_{1 \leq i \leq k} \hat{T}_i(y, w, x) ,$$

where $k \leq m$, and each *cluster* \hat{T}_i is the conjunction of some T_i 's. The conjunction scheduling problem consists of deciding the value of k , and what bit relations are merged into the i -th cluster.

Given a set of present states $S(x)$, the set of their successors $P(y)$ is the *image* of $S(x)$ and is computed by

$$P(y) = \exists x, w. S(x) \wedge \bigwedge_{1 \leq i \leq k} \hat{T}_i(y, w, x) .$$

We assume that the computation is performed as follows:

$$\exists v^1. (\hat{T}_1 \wedge \dots \wedge \exists v^k. (\hat{T}_k \wedge S)) ,$$

where v^i is the set of variables in $(x \cup w) \setminus \bigcup_{i < j \leq k} v^j$ that do not appear in $\hat{T}_1, \dots, \hat{T}_{i-1}$.

The ordering algorithm discussed in this paper is based on the analysis of the dependence matrix of the transition relation.

Definition 1. *The dependence matrix of an ordered set of functions (f_1, \dots, f_m) depending on variables x_1, \dots, x_n is a matrix D with m rows and n columns such that $d_{ij} = 1$ if function f_i depends on variable x_j , and $d_{ij} = 0$ otherwise.*

We say that Row i intersects Column j in D if $d_{ij} = 1$.

When ordering a transition relation for image computation, the rows of the dependence matrix correspond to a permutation of the conjuncts. We assume that the conjunction order is bottom-up. (That is, the subrelation corresponding to d_m is first conjoined to the set of states S . The result is then conjoined the subrelation corresponding to d_{m-1} , and so on.) If $l_j(h_j)$ is the smallest (largest) index i in column j such that $d_{ij} = 1$, respectively, then variable x_j can be quantified as soon as the subrelation corresponding to d_i is conjoined. This observation motivates the following definition.

Definition 2. Let D be a dependence matrix. The normalized average total lifetime of the variables in D is given by

$$\lambda = \frac{\sum_{1 \leq j \leq n} (m - l_j + 1)}{n \cdot m}.$$

The normalized average active lifetime of the variables in D is given by

$$\alpha = \frac{\sum_{1 \leq j \leq n} (h_j - l_j + 1)}{n \cdot m}.$$

The two quantities λ and α are related to the quality of a conjunction schedule as discussed in Section [4](#).

3 Ordering the Bit Relations

A good conjunction schedule minimizes the sizes of the intermediate BDDs during image computation. Estimation of the BDD sizes is difficult; hence, heuristic algorithms look at the number of variables in the support of a BDD as to an indicator of its size. Since in image computation, after clustering, the first conjunction involves the frontier states S , and since it is usual for S to depend on most state variables, most variables enter the computation from the start, and attention is focused on their exit time. In this context the total lifetime of a variable is the most important parameter that the conjunction schedule should minimize.

There is, however, another important factor that the quantification schedule disregards and that affects the size of BDDs and the cost of operating on them. Consider two functions $f(x_1, \dots, x_n)$ and $g(x_1, \dots, x_k)$, $k < n$, with the variable order $x_1 < \dots < x_n$. In the computation of $f \wedge g$ the recursion is never deeper than k . Even though all n variables appear in the operands, and may appear in the result, only k of them are *active*.

If the case in which f is conjoined with $h(x_k, \dots, x_n)$, a node of f labeled by one of x_1, \dots, x_{k-1} corresponds to at most one node in the result. (At most two nodes if complement arcs are used and the node is reachable from the root of f through paths of different complementation parity.) In this case also, we can conclude that some variables are not fully involved in the conjunction, and are not to be considered active. In the more general case in which some variable v of one operand does not appear in the other, but other variables both above and below v in the order do, the above observations do not hold, but fewer variables in one operand often mean a smaller BDD for that operand.

The above qualitative analysis suggests that even when all state variables appear in S it is advantageous to delay the appearance of these variables in the conjunctions. The primary input variables are not usually present in S ; hence, delaying their introduction is also beneficial. As a consequence, we propose a scheduling procedure that tries to minimize the average *active* lifetime of variables.

The overall strategy of our approach is to first order the bit relations, then cluster them, and finally order the clusters. (This is the same strategy of [\[14\]](#).) The order of the bit relations guides the clustering process. If the graph associated to the matrix has

more than one connected component, each is dealt with individually and the overall schedule is obtained by concatenation of the schedules of the components. This aspect is discussed in Section 5. Here we assume that the graph consists of one component only.

The ordering procedure is based on the algorithm of Hellerman and Rarick [3] to put a matrix in bordered block lower-triangular form. Since we are not interested in non-zero pivots, we do not need to separate the border from the lower triangular part. This leads to a slightly simplified version of the procedure originally known as *Preassigned Pivot Procedure* (P^3).

Our modified version will be called *Minimal Lifetime Permutation* (MLP). It works on a rectangular matrix. Initially all non-zero rows and columns of the matrix form the *active submatrix*. The procedure then iteratively removes rows and columns from the active submatrix and assigns them to final positions. Once the active submatrix vanishes, the matrix has been permuted. Procedure MLP consists of two phases:

- Permute singleton rows to the top and their columns to the left. (We do not permute singleton columns to the right and their rows to the bottom as in [4].) Remove the permuted rows and corresponding columns from the active submatrix.
- Iteratively choose the column that intersects the maximum number of shortest rows of the active submatrix. Move that column out of the the active submatrix and immediately to its left. If the shortest rows have length 1, they are moved out of the active submatrix and immediately above it.

Example 1. Consider the leftmost matrix of Fig. 1. We first permute Row 5 to the top

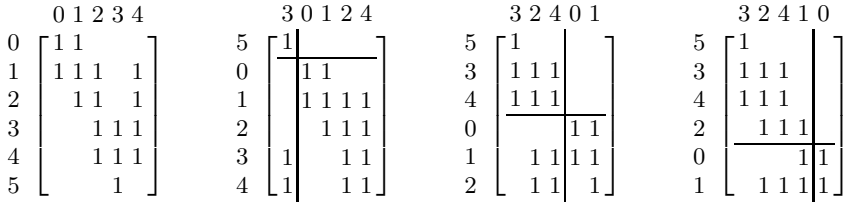


Fig. 1. Example of border-block lower triangularization

and Column 3 to the left. This gives the second matrix from the left in Fig. 1. The active submatrix now consists of Rows (0, 1, 2, 3, 4) and Columns (0, 1, 2, 4). Since there are no singleton rows in the active submatrix, we move to the second phase.

The shortest rows have length 2, and Columns 2 and 4 intersect two of them. We permute Column 2 to the second position and remove it from the active submatrix. As a result, the length of Rows 3 and 4 decreases to 1. This leads us to choose Column 4 and Rows 3 and 4, which are also permuted out of the active submatrix. The situation is depicted in the third matrix from the left in Fig. 1.

Now the active submatrix consists of Rows $(0, 1, 2)$ and Columns $(0, 1)$. The shortest row is Row 2, which has length 1, and the corresponding column is Column 1. They are permuted out of the active submatrix, giving the rightmost matrix in Fig. 1. Now the active submatrix has only one column left and the algorithm terminates. \square

Notice that the original P^3 algorithm does not pay attention to delaying the entry of the variables. Permuting the last two rows of the result of the example improves it in that respect. We have therefore modified MLP with respect to P^3 so as to reduce the average active lifetime α of the resulting matrix.

1. Unlike P^3 , we do not move column singletons to the bottom right corner of the active submatrix. Moving such columns may bring very long rows to the bottom of the matrix, causing the early entry of many variables.
2. When we move 1-length rows, we move the longest first. The length is in this case the number of ones in the entire row, not just in active region. This reduces active lifetime because it leaves shorter rows in the lower part of the matrix.

The pseudocode of the MLP procedure is shown in Fig. 2.

```

MLP( $D$ ) {
  for each (connected component  $C$  of  $D$  in aspect ratio order) {
     $C := \text{MOVE\_SINGLETON\_ROWS}(C)$ ;
     $C := \text{MOVE\_BEST\_COLUMNS}(C)$ ;
  }
}

MOVE\_SINGLETON\_ROWS( $C$ ) {
  sort rows in order of increasing length;
  while (length of first row is 1) {
    move column intersecting first row to the left;
    move the singleton rows of the leftmost column to the top of the matrix;
    update row list;
  }
}

MOVE\_BEST\_COLUMNS( $C$ ) {
   $r :=$  list of rows sorted in order of increasing length;
  while ( $r$  not empty) {
     $c :=$  column intersecting the most shortest rows;
    move  $c$  to the left of the active region;
    move the rows of length 1 intersecting  $c$  to the top of the active region;
    shrink the active region;
    update  $r$ ;
  }
}

```

Fig. 2. Pseudocode of the MLP procedure

3.1 Complexity

The dependence matrix is stored as a sparse matrix. The complexity of the MLP procedure is $O(np)$ [14], where n is the number of columns and p is the number of non-zeroes of the matrix. The dependence matrices of large circuits encountered in practice are sparse, which means that p is proportional to the number of rows of the matrix. In these cases, the algorithm runs in time linear in the size of the matrix, or quadratic in the number of state variables.

This claim is supported by the measurements reported in Table 1. The table shows, for each circuit, the number of binary state variables (FFs), the number of binary primary inputs (PIs), the percentage of non-zero elements in the dependence matrix, the average row length, the number of connected components of the graph associated to the matrix (CCs), which will be discussed in Section 4 and the time required to run the algorithm. The set of benchmark circuits includes some large ones for which reachability is not feasible, so that the performance of MLP can be observed on a larger range of input sizes. The speed of the procedure makes it suitable for use in image computation algorithms like the one of [14], which may benefit from changing the conjunction schedule dynamically several times during an image computation.

Fig 2 shows a log-log plot of time required to run the MLP algorithm as a function of the number of memory elements. The solid line has a slope of 2. One can see that, in practice, performance is indeed quadratic in the number of memory elements.

Table 1. Time for MLP

Design	FFs	PIs	Non-zeroes (%)	Avg. Row Length	CCs	Time (secs)
mult32a	32	33	55.29	35.9	1	0.001
model	61	15	12.25	7.5	1	0.001
mm30	90	67	19.51	24.0	1	0.01
s3330	132	40	3.73	5.7	1	0.001
s4863opt	154	54	5.96	8.8	5	0.001
s5378opt	156	35	5.33	6.2	2	0.001
s4863	164	51	6.36	10.2	7	0.01
s5378	179	35	4.50	8.8	5	0.001
s3384	183	43	4.61	10.3	7	0.01
fabric_str	190	38	2.11	4.0	1	0.01
cps	231	105	15.24	50.7	1	0.03
s6669	239	84	3.45	10.0	5	0.01
hw_top	356	49	21.29	32.7	1	0.04
s15850	597	15	3.62	21.3	3	0.07
s13207	669	44	0.72	4.7	11	0.05
trainflat	865	102	6.72	64.5	1	0.25
s38584	1452	14	0.84	12.1	1	0.59
s38417	1636	103	0.80	13.9	2	0.53
s35932	1728	35	0.22	4.0	1	0.89
avq	3705	28	0.22	8.2	10	1.56

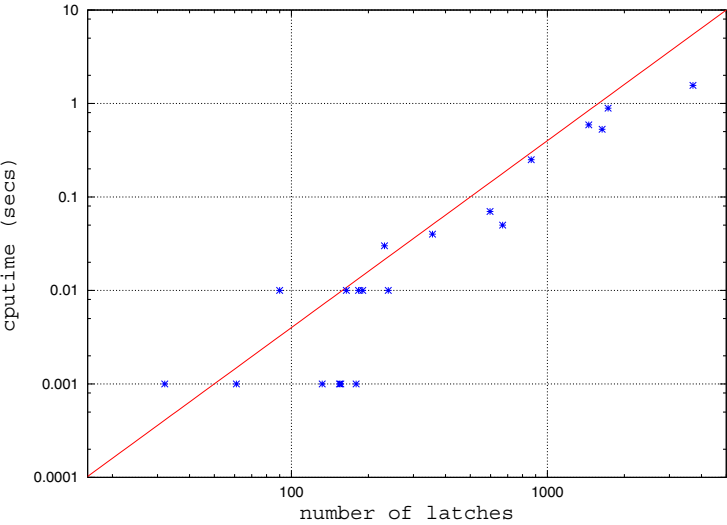


Fig. 3. Empirical complexity result of MLP

4 Clustering

A popular approach to clustering [17] is as follows. Starting from an ordered list of conjuncts, one merges the first few of them. When the size of the BDD for the merged conjuncts exceeds a given threshold, the last conjunction is discarded, and the merged conjuncts are set aside as the first cluster. The process is then repeated on the rest of the list. The greediness of this approach may lead to suboptimal results as shown by the following example.

Example 2. Consider the matrix of Fig. 1. Suppose that conjoining all bit relations leads to a BDD too large for the given resources, while two clusters can be built within the given limit. A greedy approach may group Rows 2–5 and Rows 0–1, whereas clustering Rows 3–5 and Rows 0–2 results in a smaller value of α . □

	0	1	2	3	4
0	1	1			
1	1	1	1		
2	1	1	1		
3			1	1	1
4				1	1
5				1	1

Fig. 4. Greedy clustering example

The previous example suggests that an effective clustering strategy should be based on a notion of affinity between bit relations. It is important to retain the low complexity of the greedy approach (linear number of BDD operations), because the conjunctions of the BDDs may be expensive. These observations lead to the following definition and algorithm.



Definition 3. Let d_i be the i -th row of the dependence matrix D . Let $|d|$ be the length (number of non-zero entries) of row vector d . Finally, let $d_i \cdot d_j$ designate the inner product of d_i and d_j . The affinity, β_{ij} of Rows i and j is defined as:

$$\beta_{ij} = \frac{d_i \cdot d_j}{|d_i + d_j|}.$$

Our method is to compute affinity for pairs of adjacent rows after running MLP, and then merge the pair with the highest affinity. Merging consists of conjoining the BDDs for the two rows. As in the greedy algorithm, merging is accepted only if the resulting BDD size does not exceed the cluster threshold size. If the threshold is exceeded, a *barrier* is introduced between the two rows. The process is then recursively applied to the two subsets of the rows above and below the barrier.





If the size of the conjunction BDD is below the threshold, the algorithm computes the affinity for the new row and its neighbors and then selects a new pair with highest affinity. The terminal case of the recursion is when only one row is left.

5 Connected Components

When the dependence matrix is regarded as the adjacency matrix of a directed graph, the presence of more than one connected component in the graph signals the existence of non-trivial block-diagonal form for the matrix. Multiple connected components occur rather frequently in practice. Even more so when the image computation is decomposed disjunctively . Some evidence to that effect is presented in Table .

Identifying the connected components of the graph and then ordering and clustering each component individually is desirable. Finding the connected components is linear in the size of the matrix, even for full matrices, and leads to decomposition of the problem. Clustering also benefits from decomposition, because there is no point in clustering terms with disjoint support.

However, the main reason to decompose the problem according to the connected components is to reduce the average active lifetime of variables, as shown by the following example.

Example 3. The matrix of Fig.  illustrates the benefits of partitioning based on the connected components before running MLP. The result of MLP without partitioning is shown by the middle matrix of Fig. . For this matrix, $\lambda = 0.65$, $\alpha = 0.52$. Finally, the result of MLP with partitioning is given by the rightmost matrix of Fig.  ($\lambda = 0.60$, $\alpha = 0.40$). 

The overall order for a matrix that has multiple connected components is obtained by concatenating the orders for each component. The sorting of the components is based on the ratio of number of columns to number of rows. If the ratio is large, the component goes toward the beginning of the schedule. This is to minimize variable lifetime.

	0	1	2	3	4	5	6	7
0	1	1						
1		1	1					
2	1	1	1	1	1	1		
3		1	1	1				
4	1					1	1	
5		1	1	1				

	0	2	1	3	5	6	7	4
0	1	1						
1			1	1				
3				1	1	1		
5				1	1	1		
4	1					1	1	
2	1	1			1	1	1	

	1	3	5	0	2	6	7	4
1	1	1						
3	1	1	1					
5	1	1	1					
0				1	1			
4				1		1	1	
2				1	1	1	1	1

Fig. 5. Example illustrating the benefits of connected component partitioning

6 Border Block Triangular versus Block Triangular

Proper permutation of the rows and column of a matrix can reduce the number of fill-ins during Gaussian elimination. The objective of such permutation is to put the matrix in block-triangular form. In particular, the elements on the main diagonal must be different from 0, so that they can be used as pivots. Dulmadge and Mendlesohn [10] proposed a procedure that consists of two phases: In the first phase a maximum matching between rows and columns. This matching determines a row permutation that brings element A_{ij} to the main diagonal if Row i is matched to Column j . (If a perfect matching does not exist, the matrix is singular.)

In the second phase, a symmetric permutation is applied to the result of the first phase to get a lower block triangular form with small blocks. This is achieved by identifying the strongly connected components (SCCs) of the graph associated to the matrix, and sorting the SCCs in reverse topological order.

When compared to the approach of Section 4, the approach of Dulmadge and Mendlesohn suffers from two main drawbacks. First, the constraint of having non-zero pivots is not germane to the computation of a good conjunction schedule for image computation and tends to produce suboptimal results. Second, the procedure does not address the ordering of the rows inside each SCC of the graph. In the not infrequent case when the entire graph is strongly connected, the approach is ineffective. These observations were confirmed by our implementation of the procedure of Dulmadge and Mendlesohn.

7 Experimental Results

We have implemented the conjunction scheduling algorithm in VIS 1.4 [11]. The experiments were conducted on a 400 MHz Pentium II machine with 1GB of RAM running Linux. We have applied the algorithm to two image computation methods; the one of Ranjan *et al.* [12] and the hybrid image computation of [13]. In the experiments, we set the memory usage limit size to 750MB.

Table 1 compares the runtimes and peak BDD live nodes for reachability analysis with dynamic variable reordering. The first two columns show the name and the number of latches of each circuit. The next four columns compare the runtimes and the next four columns compare the peak BDD live nodes. IWLS95 is the method in [14];

MLP is the one we present; Hybrid is the image computation method in [15] that uses IWLS95 for conjunction scheduling, and HMLP is the same hybrid method using MLP for conjunction scheduling. We compare MLP to IWLS95 and HMLP to Hybrid.

MLP wins in 19 out of 21 cases against IWLS95, often by large margins (*s1269*, *elevator*, *s1512*, *s4863opt*, *s4863*, *s3271*, *s5378opt*, *nosel*, *soap*, and *cps*). The comparison of HMLP and Hybrid gives similar results. Moreover, we can see that the peak BDD live nodes in MLP and HMLP are in most cases smaller than in IWLS95 and Hybrid, respectively. Table 2 shows that MLP improves the conjunction schedule over IWLS95 and so does HMLP over Hybrid.

The hybrid method is reported to improve over IWLS95 [15]. Table 2 confirms that. However, the gain of MLP over IWLS95 is larger than the one of HMLP over Hybrid; MLP was even faster than HMLP in 13 out of 21 cases. This might be because MLP improves the conjunction schedule; hence, it reduces the benefit of splitting in the hybrid method. HMLP may require re-calibration of the parameters such as lifetime threshold and minimum and maximum splitting depth in the hybrid method [15].

Table 2. Comparison with reachability analysis

Circuit	FF	Time (seconds)				Peak live nodes (K)			
		IWLS95	MLP	Hybrid	HMLP	IWLS95	MLP	Hybrid	HMLP
sbc	28	11.4	6.3	11.3	6.9	21.9	20.8	23.4	22.6
clma	33	12.5	7.7	26.3	19.4	30.4	30.4	39.3	30.4
clmb	33	10.6	7.1	35.1	19.8	35.3	35.3	45.6	35.3
bpb	36	531.5	500.1	433.7	41.6	162.0	311.1	154.9	67.2
s1269	37	6428.7	1910.9	1788.7	1657.8	3652.3	1359.7	2045.0	1413.9
elevator	50	1633.1	351.3	577.8	505.2	1237.3	440.4	498.8	404.5
s1512	57	2304.1	599.2	1882.6	659.4	189.9	136.5	178.8	99.6
model	61	10.6	6.2	11.5	8.3	23.1	13.9	23.0	18.2
s4863opt	88	13736.6	891.4	1775.0	710.9	896.2	403.3	931.4	504.6
mm30	90	56.2	29.8	50.5	13.0	81.2	75.3	91.8	65.1
reactor	91	38.1	38.3	59.4	58.3	40.1	43.4	50.9	45.3
s4863	104	Time-out	2805.6	Time-out	1586.7	Time-out	1374.0	Time-out	1058.7
s3271	116	67885.9	3174.1	6069.6	2604.3	975.9	999.4	1509.9	1094.4
s5378opt	121	8584.4	2201.1	5243.3	3804.0	743.2	572.2	715.5	498.7
nosel	128	5861.8	56.0	3068.2	64.4	2584.0	58.0	818.6	62.5
s3330	132	13380.3	15327.8	5034.6	5377.3	11072.3	13062.2	8778.0	9380.0
cps1364opt	137	965.8	762.4	638.0	582.5	499.6	444.9	333.5	233.0
soap	140	137.7	34.3	172.5	116.9	139.5	26.8	155.7	85.1
dsip	224	237.8	199.4	332.6	225.8	42.3	64.2	135.6	95.8
cps	231	5186.4	599.8	2841.5	1050.7	1626.1	342.3	1039.8	364.1
sfeistel	293	950.1	829.2	921.2	1460.4	228.6	159.5	203.8	260.6

Table 3 shows runtimes and peak BDD live nodes with fixed variable orders for reachability analysis. In this table, we compare MLP versus IWLS95 using two fixed

variable orders saved at the end of two reachability analysis experiments run with dynamic reordering: one with MLP and the other with IWLS.

Comparing the results obtained with dynamic reordering has the drawback that reordering is effectively a large source of noise. On the other hand, reordering is used in practice in many cases, and it is important to study the performance of an algorithm in these conditions. Moreover, in the case of our comparison, there tends to be a correlation between a fixed order obtained at the end of a reachability experiment and the method used for conjunction scheduling in the experiment itself. This correlation is effectively absent from experiments run with dynamic reordering. By reporting data obtained with and without dynamic reordering, we show that the gains due to the improved conjunction scheduling are not greatly affected by the ordering strategy.

In Table 4, MLP wins over IWLS95 in 17 cases out of 21 examples, often by large margins. The same happens in the comparison of HMLP to Hybrid.

Table 3. Reachability comparison with fixed orders

Circuit	FF	Time (seconds)				Peak live nodes (K)			
		Fixed from MLP		Fixed from IWLS95		Fixed from MLP		Fixed from IWLS95	
		IWLS95	MLP	IWLS95	MLP	IWLS95	MLP	IWLS95	MLP
sbc	28	3.3	1.5	2.7	1.7	317.3	206.8	365.5	238.7
clma	33	23.0	7.6	22.6	7.6	221.6	96.3	221.6	96.3
clmb	33	8.8	7.0	8.4	6.9	211.9	108.0	211.9	108.0
bpb	36	334.5	4.0	600.2	3.35	1742.1	124.3	1447.5	113.6
s1269	37	177.9	702.8	359.4	282.8	9000.0	14001.4	26507.5	25565.0
elevator	50	173.2	96.4	319.9	304.2	5243.4	5723.1	10984.0	9549.0
s1512	57	409.9	298.4	413.5	395.4	2789.5	2493.3	6680.0	5257.3
model	61	0.9	0.8	1.9	1.9	19.9	19.9	50.2	47.1
s4863opt	88	1298.2	1337.3	5224.7	104.9	7031.2	7397.5	25383.0	682.3
mm30	90	0.8	1.2	0.5	1.2	96.5	87.6	67.0	84.9
reactor	91	16.4	14.7	13.3	13.2	153.8	94.7	124.3	95.7
s3271	116	3899.9	199.3	482.2	Time-out	6040.6	2887.9	3386.0	Time-out
s5378opt	121	765.8	159.1	12293.1	903.4	21691.4	6887.7	23363.0	4338.2
nosel	128	36.8	17.1	1983.8	804.9	296.2	108.8	1091.8	3758.1
s3330	132	174.1	200.3	Time-out	20746.0	10500.1	13005.3	Time-out	23701.6
cps1364opt	137	47.3	38.9	68.6	37.8	1357.3	1513.7	1813.9	1205.2
soap	140	141.1	105.9	46.0	36.2	3100.8	3089.7	968.2	750.7
dsip	224	51.6	4.8	10.5	10.9	4509.8	376.3	852.9	659.1
cps	231	6032.1	768.6	89.6	39.7	25213.8	8191.8	4940.0	1437.9
sfeistel	293	10.2	8.7	Mem-out	Mem-out	744.9	744.8	Mem-out	Mem-out

Table 4 compares total lifetimes and active lifetimes between IWLS95 and MLP. The lifetimes in the second to fourth columns were computed before clustering and after ordering. The lifetimes in the next four columns were obtained after clustering. The total lifetime is related to how good a quantification schedule is; the active lifetime says how good the conjunction schedule is. The lifetime before clustering shows how

good ordering was, whereas the lifetime after clustering shows how good clustering was with the given order. Since we use partitioned transition relation based on this clustering for conjunctions, a short lifetime after clustering is more important. However, a short lifetime before clustering is also important, in that it provides better initial condition to clustering.

Table 4 shows that MLP gives smaller values in both total and active lifetime as well as in both before and after clustering in most cases. The differences in active lifetime between IWLS95 and MLP is typically larger than the one in total lifetime. In terms of active lifetime after clustering, MLP wins 15 cases and ties two more out of 21 designs. In the case of *nosel*, even though MLP has a larger active lifetime, MLP leads to much faster reachability than IWLS95. This will be explained in the discussion of Table 5.

Table 4. Comparison of variable lifetime

Circuit	Before clustering				After clustering			
	Total lifetime		Active lifetime		Total lifetime		Active lifetime	
	IWLS95	MLP	IWLS95	MLP	IWLS95	MLP	IWLS95	MLP
sbc	0.53	0.49	0.29	0.22	0.51	0.29	0.48	0.24
clma	0.08	0.08	0.07	0.07	0.10	0.09	0.10	0.08
clmb	0.08	0.08	0.08	0.08	0.10	0.09	0.10	0.09
bpb	0.55	0.54	0.37	0.40	0.53	0.58	0.51	0.56
s1269	0.53	0.46	0.46	0.40	0.71	0.64	0.60	0.57
elevator	0.47	0.41	0.42	0.26	0.53	0.41	0.50	0.35
s1512	0.44	0.43	0.32	0.26	0.52	0.52	0.52	0.42
model	0.56	0.48	0.38	0.19	0.68	0.44	0.67	0.30
s4863opt	0.46	0.37	0.21	0.09	0.27	0.25	0.18	0.13
mm30	0.42	0.42	0.29	0.32	0.55	0.54	0.48	0.48
reactor	0.60	0.56	0.43	0.40	0.57	0.56	0.46	0.50
s4863	0.45	0.38	0.20	0.07	0.28	0.27	0.18	0.11
s3271	0.57	0.52	0.23	0.23	0.53	0.36	0.33	0.22
s5378opt	0.36	0.37	0.20	0.11	0.37	0.24	0.28	0.12
nosel	0.52	0.52	0.06	0.05	0.32	0.40	0.12	0.25
s3330	0.15	0.15	0.12	0.11	0.32	0.28	0.24	0.21
cps1364opt	0.71	0.67	0.60	0.52	0.73	0.65	0.61	0.53
soap	0.64	0.58	0.44	0.40	0.58	0.55	0.51	0.39
dsip	0.52	0.52	0.05	0.04	0.31	0.27	0.08	0.08
cps	0.64	0.54	0.38	0.30	0.58	0.54	0.41	0.33
sfeistel	0.53	0.62	0.25	0.45	0.52	0.56	0.38	0.41

Fig. 6 represents the log-log scale plot of the improvements in runtime (from Table 4) versus the improvements in active lifetime (from Table 4). Even though the points are quite scattered, runtime and active lifetime appear to be related.

Table 5 reports the number of clusters and the number of variables in the partitioned transition relation after clustering. The MLP algorithm does not try to minimize these numbers; instead it tries to minimize active lifetime. However, these numbers are also good indicators for how good clustering is: Both fewer clusters and fewer variables

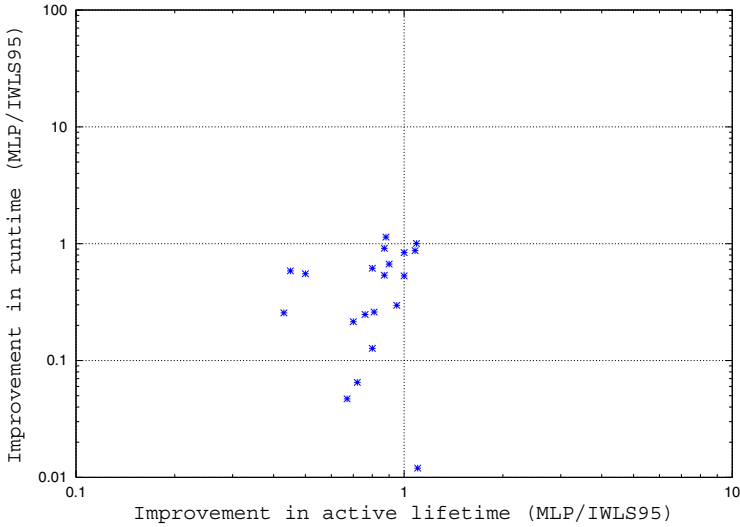


Fig. 6. Runtime versus active lifetime. Experiments are run with IWLS95 and MLP. Data for *s4863* is not included because reachability only completed with MLP

are in general beneficial for conjunctions. In terms of the number of clusters, IWLS95 wins 12 cases and MLP wins 6 cases, whereas in terms of the number of variables, MLP wins 16 cases and IWLS95 wins only 1 case. This tells us that IWLS95 tends to produce fewer clusters, whereas MLP tends to have fewer variables from clustering. The problem of *nosel* in Table 4 can be explained by the number of clusters. In this case, IWLS95 has 9 clusters, whereas MLP has only 3 clusters. In general, though, it should be possible to improve MLP to have less clusters.

Fig. 4 shows the dependence matrix of *s4863* after clustering from IWLS95, while Fig. 5 shows the matrix from MLP. These two dependence matrices and their lifetimes illustrate why MLP often performs better than IWLS95. Even though their total lifetimes are similar ($\lambda(\text{IWLS95})=0.28$ and $\lambda(\text{MLP})=0.27$), the active lifetime of MLP ($\alpha=0.11$) is much smaller than the one of IWLS95 ($\alpha=0.18$). This difference helps conjunction schedule by introducing active variables as late as possible.

Fig. 6 is the dependence matrix from the block triangularization procedure discussed in Section 4. The rows of zeroes used to pad the matrix to make it square have been removed in the figure. One can see that this matrix is inferior to those of Figures 4 and 5.

8 Conclusions

We have presented an algorithm for the conjunction scheduling of a transition relation in image computation. The algorithm has low complexity and can be used dynamically by image computation algorithms that change the dependence matrix repeatedly [14]. Our

Table 5. Comparison of clustering results

Circuit	Number of clusters		Number of variables	
	IWLS95	MLP	IWLS95	MLP
sbc	2	3	41	35
clma	3	4	45	43
clmb	5	6	46	45
bpb	2	2	41	39
s1269	5	7	51	51
elevator	5	3	53	52
s1512	2	2	60	58
model	2	3	61	61
s4863opt	40	35	112	102
mm30	6	3	119	118
reactor	6	6	91	91
s4863	39	38	117	118
s3271	8	13	136	116
s5378opt	6	8	114	104
nosel	9	3	129	128
s3330	7	8	138	141
cps1364opt	25	26	205	205
soap	6	8	141	141
dsip	17	18	260	249
cps	45	41	304	303
sfeistel	27	42	319	361

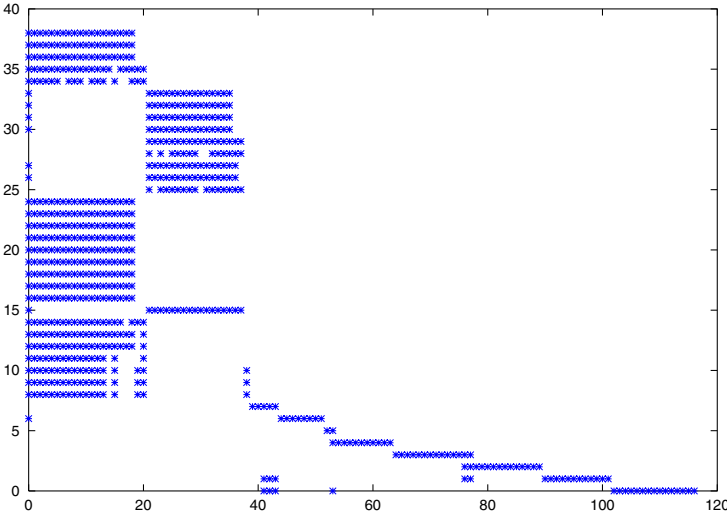


Fig. 7. Dependence matrix of *s4863* with IWLS95 after clustering

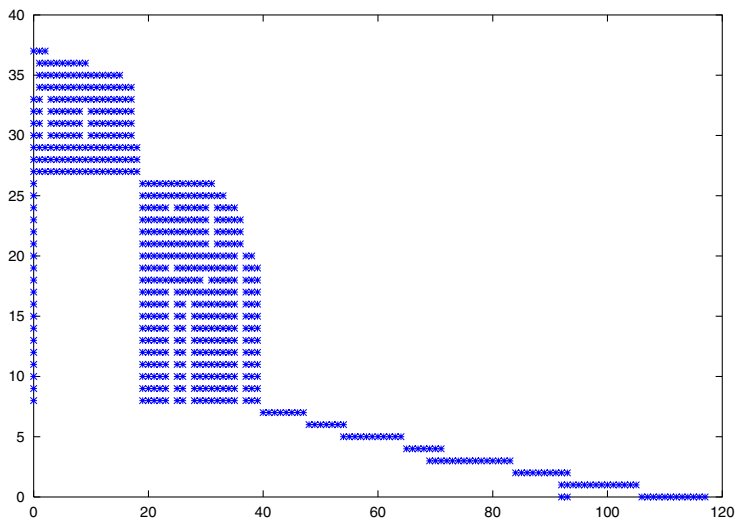


Fig. 8. Dependence matrix of *s4863* with MLP after clustering

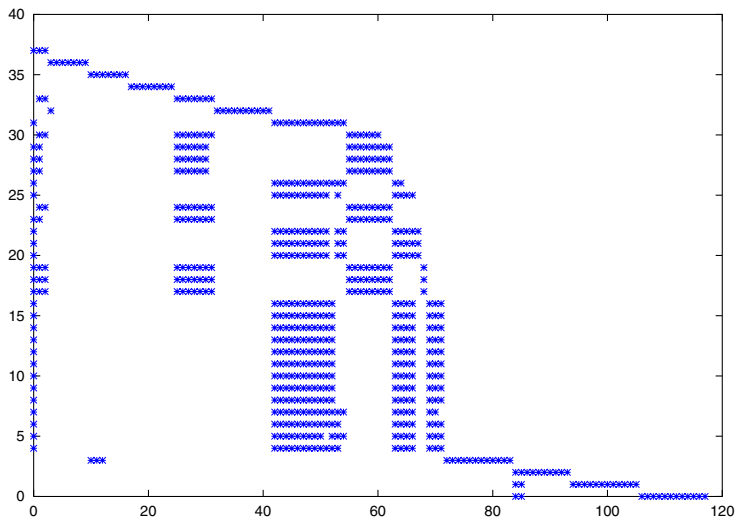


Fig. 9. Dependence matrix of *s4863* with block triangularization

experiments indicate that our approach outperforms the best method published so far. We have identified two parameters that characterize the quality of a schedule: the total lifetime, and the active lifetime of the variables. These parameters reflect the number of conjunction steps in which variables participate, and decreases in their values correlate to decreases in image computation run times.

We are investigating possible improvements to the algorithm for image conjunction scheduling so that cases like *sfeistel* from Section 4 are handled more efficiently. The extension of the proposed method to preimage computation is left as future work. One important difference between image and preimage computations in the MLP algorithm is the following. The dependence matrix we consider contains only x and w variables. This is because considering the y variables does not help for conjunction scheduling, while it may double the problem size. In image computation, we want to quantify both x and w variables as early as possible. In preimage computation, by contrast, we want to quantify the w variables as early as possible, whereas we want to introduce the x variables as late as possible in the schedule. Therefore, conjunction scheduling for image computation can be done with one MLP problem with x and w variables, whereas for preimage computation, we need to solve two MLP problems at the same time (one for the x and the other for the w variables) to reduce the overall variable lifetime [10].

The affinity-based clustering procedure is simple and effective. However, there are several alternative approaches that should be investigated. For instance, one may want to target directly the maximization of the quantified variables. That is, one may want to cluster so as to quantify as many variables as possible before image computation starts.

References

- [1] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.
- [2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [3] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 403–407, San Francisco, CA, June 1991.
- [4] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 134–137, November 1990.
- [5] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.
- [6] A. L. Dulmadge and N. S. Mendelsohn. Two algorithms for bipartite graphs. *J. Soc. Indust. Appl. Math.*, 11:183–193, March 1963.
- [7] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818.
- [8] E. Hellerman and D. C. Rarick. The partitioned preassigned pivot procedure (P^4). In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 67–76. Plenum Press, New York, 1972.

- [9] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *Proceedings of the International Conference on Computer Design*, pages 12–19, Austin, TX, October 1996.
- [10] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for FSM traversal. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 476–479, Santa Clara, CA, November 1991.
- [11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
- [12] I.-H. Moon. *Efficient Reachability Algorithms in Symbolic Model Checking*. PhD thesis, University of Colorado at Boulder, Department of Electrical and Computer Engineering, July 2000.
- [13] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the Design Automation Conference*, pages 23–28, Los Angeles, CA, June 2000.
- [14] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA., May 1995.
- [15] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990.

B2M: A Semantic Based Tool for BLIF Hardware Descriptions

David Basin, Stefan Friedrich, and Sebastian Mödersheim

Institute for Computer Science, University of Freiburg, Germany
{basin,friedric,moedersh}@informatik.uni-freiburg.de

Abstract. BLIF is a hardware description language designed for the hierarchical description of sequential circuits. We give a denotational semantics for BLIF-MV, a popular dialect of BLIF, that interprets hardware descriptions in WS1S, the weak monadic second-order logic of one successor. We show how, using a decision procedure for WS1S, our semantics provides a simple but effective basis for diverse kinds of symbolic reasoning about circuit descriptions, including simulation, equivalence testing, and the automatic verification of safety properties. We illustrate these ideas with the B2M tool, which compiles circuit descriptions down to WS1S formulae and analyzes them using the MONA system.

1 Introduction

BLIF (Berkeley Logic Interchange Format) is a hardware description language designed for the hierarchical description of sequential circuits, which serves as an interchange format for synthesis and verification tools. To better support verification, BLIF was later modified and extended to BLIF-MV (BLIF multi-value [7]), which we will consider in this article.

For building simulation, synthesis, and verification tools that interpret BLIF-MV, it is important that the language has a well-defined semantics. The currently defined semantics [10, 13] are operational: The latches define a state that is continually updated by the combinational logic. In this paper we give an alternative, denotational, semantics for BLIF-MV that provides a formal basis for automating analysis of BLIF-MV circuit specifications using “off-the-shelf” decision procedures. We interpret BLIF-MV specifications as formulae in WS1S, the weak monadic second-order logic of one successor. This logic is decidable and the MONA-system [6] implements a decision procedure for it. We have built a compiler, B2M, that translates BLIF-MV specifications into the input language of MONA and provide in this way a powerful environment for different kinds of symbolic reasoning about BLIF-MV.

Our intention here is not to compete with state-of-the-art verification systems like VIS [14], which incorporate many specialized and highly tuned algorithms for building automata from BLIF-MV specifications. Instead, we see our contributions at the level of semantics for hardware description languages; our goal is to provide semantic based methodologies for prototyping and building analysis tools for these languages. We expand on these points below.

On the semantic side, we interpret circuit descriptions logically in the monadic logics S1S and WS1S as statements about the evolution of signals over time. We use two logics for pragmatic reasons: S1S gives a simple reading of circuits operating over signals over infinite time intervals, which we then recast in WS1S, where signals range over finite time intervals, in order to use existing decision procedures.

This approach is interesting for several reasons. First, the semantic explanations we give are denotational: the meaning of a circuit is built from the meaning of its parts. As these monadic logics have simple set-theoretic semantics, so do our denotations. This provides simple alternative accounts of BLIF-MV (over both infinite and finite time intervals) that are helpful in the same way that a declarative semantics of a language (e.g., the least fixedpoint semantics of Prolog) complements an operational one (SLD-resolution). Second, our semantics also have an operational side, which comes at no extra cost. Monadic logics like WS1S are decided using automata-theoretic techniques: every formula is equivalent to an automaton that describes the models of the formula. Hence, the decision procedure for WS1S, which builds automata from formulae, guarantees that there is an agreement between these two semantics. Finally, our use of monadic logic has some generality in that it can be used to formalize (regular fragments of) other hardware description languages in a way suitable for prototyping them and for experimenting with existing automated reasoning tools. For instance, a large subset of VERILOG can be translated to BLIF-MV.

On the tool side, we show how our semantics can be used to automate reasoning about BLIF-MV specifications.¹ Namely, the formulae output by our compiler can be input to the MONA system and subjected to various kinds of analysis. For example, we use MONA to produce a minimal finite-state representation of the circuits, which can be used for simulation. Alternatively, we can automatically verify (or find counter-examples for) equivalence between circuits, or check safety properties. For simulation and formal analysis, the close connection between the logical and the operational side makes our approach particularly flexible since both inputs and outputs of the circuit can easily be restricted to cases of interest by formulating appropriate constraints in WS1S.

Although we do use a general purpose system for these tasks, MONA is highly optimized and uses BDD-based algorithms to represent and manipulate automata. However the cost of this generality is that the conversion from a BLIF-MV description to an automaton is slower than state-of-the-art synthesis systems like VIS; still our approach produces acceptable run-times on many realistic examples. Moreover, by avoiding specialized algorithms and using general purpose tools, alternative symbolic manipulation procedures developed for WS1S, for example SAT-based approaches to counter-example generation [2], can easily be integrated in our work.

¹ Note that our denotational approach also supports *interactive* reasoning. We can directly reason about the formulae interpreting BLIF-MV circuits in an appropriate theory; see [3] for examples of such reasoning.

Organization. The remainder of this article is organized as follows: In Section 2, we summarize BLIF-MV and the logics S1S and WS1S. For the sake of simplicity, we restrict ourselves to the essential constructs of BLIF-MV, contained in the sublanguage Core-BLIF. In Section 3, we formalize the semantics for Core-BLIF in terms of S1S and explain how to interpret the result in WS1S. In Section 4, we show how to use the MONA system to perform different kinds of analysis on our translations. In the final section, we draw conclusions and discuss future work.

2 Background

2.1 BLIF-MV

BLIF-MV is a kind of hardware assembly language where circuits are described as directed graphs of combinational gates and sequential elements. It was developed as an extension of BLIF (dropping timing-related constructs) to serve as an interchange format for verification and simulation tools like VIS [14].

Core-BLIF. In this article we will restrict ourselves to a fragment of BLIF-MV, which we refer to as Core-BLIF. The fragment simplifies our presentation, but all constructs of BLIF-MV can be expressed in it.² The syntax is summarized in Figure 1 in an extended BNF-like notation and is explained on a simple example.

Consider a traffic light system for a pedestrian crossing that consists of two traffic lights (for pedestrians and for cars) and a button. By default, the cars have green. If a pedestrian presses the button, his light turns green (and the car's light turns red) after one time unit. If the pedestrian's light is already green then pressing the button has no effect.

In Figure 2 we give the Core-BLIF specification of the system. A system specification consists of multiple *model definitions*. For this system, we have two: one for the control logic and one for the lights. Let us begin with the control logic, which computes a function of the present signal for the cars (which is either 0 for red or 1 for green) and the state of the button; the result is the signal for the cars in the next time unit.

The control logic is given by a model definition, which has five arguments: the first argument names the circuit; the second is the list of input signals; the third is the list of output signals; the fourth (here empty) denotes the local signals (which are neither input nor output); and the fifth is the list of components from which the circuit is built. In our example, the circuit consists of only one component, a combinational gate. The combinational gate consists of an input list, an output list, an optional default output row (here 1), and a table that describes a relation between inputs and outputs. In the abstract syntax, the

² For example, multi-value signals of BLIF-MV can be encoded in Core-BLIF using binary-valued signals, since any signal over a domain of size n can be encoded by $\lceil \log_2 n \rceil$ binary signals.

```

start = model*
model = identifier × identifier* × identifier* × identifier* × component*
component = Comb(comb_gate)|Latch(latch)|Subckt(subcircuit)|Reset(comb_gate)
comb_gate = identifier* × identifier* × [row] × table
table = (row × row)*
row = literal*
literal = 0|1|DontCare
latch = identifier × identifier
subcircuit = identifier × form_act
form_act = (identifier × identifier)*

```

Fig. 1. The abstract syntax of Core-BLIF.

<pre> .model ControlLogic .inputs PresentSig Button .outputs NextSig .names PresentSig Button -> NextSig .def 1 1 1 0 .end .model Lights .inputs Button .outputs CarSig PedestSig .subckt ControlLogic PresentSig=CarSig Button=Button NextSig=Tmp .latch Tmp CarSig .names CarSig -> PedestSig 0 1 1 0 .end </pre>	<pre> (ControlLogic, [PresentSig, Button], [NextSig], [], [Comb([PresentSig, Button], [NextSig], [1], [[1, 1], [0]])])) (Lights [Button], [CarSig, PedestSig], [Tmp], [Subckt(ControlLogic, [(PresentSig, CarSig), (Button, Button), (NextSig, Tmp)]), Latch(Tmp, CarSig), Comb([CarSig], [PedSig], -, [[0], [1]], ([1], [0]])])) </pre>
(a) Concrete Syntax	(b) Abstract Syntax

Fig. 2. A simple traffic light system.

table rows are split into two parts, the input and the output pattern (e.g. the lists $[1, 1]$ and $[0]$ in the single table row of the example). The table is to be read as the disjunction of the row pairs, which themselves denote the conjunction of their literals. The default output is chosen if none of the input rows match the present value of the input signals. Hence, the given example describes the *nand* relation

$$\{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 0)\} .$$

The traffic light is constructed by using the control logic model as a subcircuit along with a latch and another combinational gate that computes the pedestrian signal from the car signal. The subcircuit call consists of the name of the called

circuit along with a mapping from formal parameters (the input/output signals of the called circuit) to actual ones (the signals in the calling circuit). The latch has only a single input and output. The initial value of a latch can be specified by reset tables. With a reset table, one specifies which combinations of initial values for the latches of the circuit are allowed. Reset tables are identical to normal tables, except for the fact that the relation holds only for the initial time point. For instance, assume we have two latches with outputs A and B and we want to specify that the initial value of the latches is either both 0 or both 1. We can specify this by the following reset table:

```
.reset A B
1 1
0 0
```

There are additional restrictions on circuits that are straightforwardly formalized outside of the grammar given. For example, each table row pair must have exactly (n, m) elements if the gate has n inputs and m outputs. Moreover, every signal (except inputs) must be the output of one unique component. Finally, there must be no combinational cycles (i.e., a cycle in the component-graph, where all components in the cycle are combinational gates) and no cycles in the dependency graph that results from the subcircuit calls.

A circuit specification has the following operational semantics. Assume the existence of a global system clock. At each clock tick the latches update their values, i.e., they take the value of the incoming signal. The new values appear at the latch outputs immediately and are propagated through all combinational parts of the circuit (i.e., the propagation stops when it reaches another latch) until a stable condition is reached. This whole propagation process happens immediately, as if all combinational gates switched without any time delay [13].

2.2 Second-Order Monadic Logics of One Successor

We now briefly describe the syntax and semantics of the second-order monadic logic of one successor S1S and its “weak” restriction WS1S. For more on these logics, see [11, 12].

Syntax. Let x and X range over disjoint sets \mathcal{V}_1 and \mathcal{V}_2 of first and second-order variables. The language of both S1S and WS1S is described by the following grammar.

$$\begin{aligned} \mathcal{T} &::= x \mid 0 \mid s(\mathcal{T}) \\ \phi &::= X(\mathcal{T}) \mid \phi \wedge \phi \mid \neg \phi \mid \exists^1 x. \phi \mid \exists^2 X. \phi \end{aligned}$$

Hence terms are built from first-order variables, the constant 0, and the successor symbol. Formulae are built from atoms $X(t)$ and are closed under conjunction, negation, and quantification over first and second-order variables.

Other connectives and quantifiers can be defined using standard classical equivalences, e.g., $\forall^1 x. \phi \equiv \neg \exists^1 x. \neg \phi$. We will also make use of various other kinds of definitional sugaring, e.g. writing 1 for $s(0)$ and using definable operators like $=$ and $<$.

Semantics. S1S formulae are interpreted in \mathbb{N} . 0 and s denote zero and the successor function, and $X(t)$ is true if the number denoted by t is in the set of numbers denoted by X . First-order quantification is quantification over natural numbers, whereas second-order quantification is quantification over sets of natural numbers. The semantics of WS1S is identical, except for the fact that second-order variables are interpreted over *finite* sets of natural numbers. Hence the formula $\forall^1 t. X(t)$ is satisfiable in S1S, but unsatisfiable in WS1S, as there is no finite set containing all natural numbers.

Although these are logics of numbers and sets, they can be viewed as logics over strings: For WS1S, any finite string $b(0)b(1)\dots b(m)$ over \mathbb{B} encodes a finite set of positions, namely $\{p \in \{0, \dots, m\} \mid b(p) = 1\}$. More generally, we can encode n strings over \mathbb{B} as a single string over \mathbb{B}^n . Hence, if $\phi(\overline{X})$ is a WS1S formula whose free second-order variables are $\overline{X} \equiv X_1, \dots, X_n$, a WS1S interpretation can be encoded by a finite string over the alphabet \mathbb{B}^n . The same holds for S1S, except that strings are *infinite*.

As a simple example, the formula ϕ given by $\forall^1 t. X(t) \leftrightarrow Y(s(t))$ states that every number in the set Y that is greater than 0 is the successor of a number in the set X and vice versa. A WS1S interpretation for this formula can be encoded by a string $\overline{b}(0)\overline{b}(1)\dots\overline{b}(m)$, where each $\overline{b}(i)$ is a letter in \mathbb{B}^2 . To visualize this, we write letters (b_1, b_2) vertically; the first track encoded in the string determines an interpretation for X , and the second an interpretation for Y . Two such interpretations for WS1S are

$$I_1 = \begin{array}{c} X \\ Y \end{array} \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 0 & 0 \\ \hline \end{array} \quad \text{and} \quad I_2 = \begin{array}{c} X \\ Y \end{array} \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}.$$

The first interpretation, for example, interprets X as $\{0, 2\}$ and Y as $\{0, 1, 3\}$. ϕ is satisfied for WS1S in the first interpretation (i.e., it is a *model* of ϕ) and not satisfied in the second, which we write as $I_1 \models_{\text{WS1S}} \phi$ and, respectively, $I_2 \not\models_{\text{WS1S}} \phi$. The same applies to S1S, if the strings are infinitely extended with 0s. Interpreted over (bit) strings, ϕ says that Y is the string X right-shifted one position, with the initial bit arbitrarily filled.

Tool Support. There are several implementations of decision procedures for WS1S [6, 9]. These are based on the fact that WS1S captures precisely the regular languages: the language associated with each WS1S formula ϕ (i.e., the set of all strings that encode a model) is regular, and vice versa. Hence, using automata theoretic techniques, given a formula ϕ , the systems construct an automaton that recognizes the models of ϕ . A formula $\phi(\overline{X})$ is a tautology iff

the corresponding automaton accepts the universal language on \mathbb{B}^n . To decide S1S a similar procedure based on Büchi automata can be used.

The decision problem for both logics is non-elementary [8]. In the case of WS1S, despite such a poor worst-case complexity, the implemented decision procedures work surprisingly well on many non-trivial problems. In particular, the MONA system has been highly optimized and can quickly process many large formulae (e.g., formulae with hundreds of thousands of symbols). The system has been used to formalize and reason about sequential hardware [4] and protocols [6], where a finite string encodes values of signals or the evolution of the system state over time. In contrast to WS1S, there is no satisfactory tool support for S1S and due to technical difficulties (concerning minimization and complementation) it seems unlikely that similarly effective tools are possible for this logic.

3 A Semantics of Core-BLIF

3.1 An S1S Semantics

In Section 2.1 we explained the informal “synchronous hardware” semantics for Core-BLIF: Combinational gates switch without delay and latches load in the current value of the input signal at every tick of the global system clock. This semantics is equivalent to stating that latches delay the incoming signal for one time unit, if we take the time between two clock ticks to be one time unit. As this delay is the only time-relevant issue that must be modeled, the natural numbers can serve as the set of time points, which can be modeled using first-order variables in S1S. Further, since a signal in Core-BLIF is a binary valued function of time, a signal can be modeled in S1S by a second-order variable, used to formalize the set of time points at which the signal has the value 1.

We now define the semantics of Core-BLIF as a family of functions $\llbracket \cdot \rrbracket_{S1S}^{NT}$, where each function maps the language associated with a non-terminal symbol NT of the Core-BLIF abstract syntax (see Figure 1) to an S1S formula. The semantics is summarized in Figure 3 and we describe below the main semantic functions.³ Figure 4 gives the S1S formulae resulting from this translation for the traffic lights example.

The function $\llbracket \cdot \rrbracket_{S1S}^{model}$ translates a circuit into a predicate definition in S1S.⁴ This way of modeling circuits as predicates (semantically, relations) is standard in higher-order logics [5]. A predicate describes a relation between input and output signals. Components of the circuit are modeled as constraints on the signals of the circuit and are conjoined together. Internal signals are hidden

³ Note that in Figure 3, quantification over a list of variables represents quantification over all members of the list.

⁴ Predicate definitions are not part of the monadic logics we defined. We can simply view them as extra-logical definitions or macros. Our use of them has a practical advantage: The MONA system supports predicate definitions and predicates are compiled individually, only once, into automata. Hence, these definitions support not only a compositional semantics, but also the hierarchical construction of automata.

Types: (\mathcal{T} is the set of S1S terms, cf. grammar in Section 2.2)

$$\llbracket \cdot \rrbracket_{S1S}^{model} : model \rightarrow \phi$$

$$\llbracket \cdot \rrbracket_{S1S}^{comp} : component \rightarrow \mathcal{T} \rightarrow \phi$$

$$\llbracket \cdot \rrbracket_{S1S}^{row} : row \rightarrow identifier^* \rightarrow \mathcal{T} \rightarrow \phi$$

$$\llbracket \cdot \rrbracket_{S1S}^{literal} : literal \rightarrow identifier \rightarrow \mathcal{T} \rightarrow \phi$$

Definitions:

$$\begin{aligned} \llbracket (name, Ins, Outs, Locals, [comp_1, \dots, comp_n]) \rrbracket_{S1S}^{model} = \\ name(Ins, Outs) \equiv \exists^2 Locals. \bigwedge_{i=1}^n \forall^1 t. \llbracket comp_i \rrbracket_{S1S}^{comp}(t) \\ \llbracket Comb(Ins, Outs, -, [(In_1, Out_1), \dots, (In_n, Out_n)]) \rrbracket_{S1S}^{comp}(t) = \\ \bigvee_{i=1}^n (\llbracket In_i \rrbracket_{S1S}^{row}(Ins)(t) \wedge \llbracket Out_i \rrbracket_{S1S}^{row}(Outs)(t)) \\ \llbracket Comb(Ins, Outs, default, [(In_1, Out_1), \dots, (In_n, Out_n)]) \rrbracket_{S1S}^{comp}(t) = \\ \bigvee_{i=1}^n (\llbracket In_i \rrbracket_{S1S}^{row}(Ins)(t) \wedge \llbracket Out_i \rrbracket_{S1S}^{row}(Outs)(t)) \\ \vee (\bigwedge_{i=1}^n \neg \llbracket In_i \rrbracket_{S1S}^{row}(Ins)(t) \wedge \llbracket default \rrbracket_{S1S}^{row}(Outs)(t)) \\ \llbracket Latch(In, Out) \rrbracket_{S1S}^{comp}(t) = In(t) \leftrightarrow Out(s(t)) \\ \llbracket Subckt(name, form_act) \rrbracket_{S1S}^{comp}(t) = name([form_act]) \\ \llbracket Reset(comb_gate) \rrbracket_{S1S}^{comp}(t) = \llbracket Comb(comb_gate) \rrbracket_{S1S}^{comp}(0) \\ \llbracket (lit_1, \dots, lit_n) \rrbracket_{S1S}^{row}(Id_1, \dots, Id_n)(t) = \bigwedge_{i=1}^n \llbracket lit_i \rrbracket_{S1S}^{literal}(Id_i)(t) \\ \llbracket 0 \rrbracket_{S1S}^{literal}(Id)(t) = \neg Id(t) \\ \llbracket 1 \rrbracket_{S1S}^{literal}(Id)(t) = Id(t) \\ \llbracket - \rrbracket_{S1S}^{literal}(Id)(t) = \text{true} \end{aligned}$$

Fig. 3. The S1S semantics of Core-BLIF.

by existential quantification, which asserts the existence of intermediate values, consistent with the constraints. The formula that models the complete circuit therefore states that all these constraints must be met at every time point. The time point is an additional parameter for the semantics of *component*, *row*, and *literal*; it can be any value of \mathcal{T} , which denotes the set of all first-order S1S terms given by the grammar in Section 2.2.

The function $\llbracket \cdot \rrbracket_{S1S}^{comp}$ is used to translate combinational gates, latches, and resets. Because combinational gates have no delay and no internal state, they can be modeled as a relation that has to hold of the corresponding signals at each time point. If no default output is given, this relation is simply the disjunction of the row pairs in the table; otherwise the relation additionally contains each input pattern that is not covered by the table together with the default value

$$\begin{aligned}
&\text{ControlLogic}(\text{PresentSig}, \text{Button}, \text{NextSig}) \equiv \\
&\quad \forall^1 t. (\text{PresentSig}(t) \wedge \text{Button}(t) \wedge \neg \text{NextSig}(t)) \vee \\
&\quad \quad (\neg(\text{PresentSig}(t) \wedge \text{Button}(t)) \wedge \text{NextSig}(t)) \\
&\text{Lights}(\text{Button}, \text{CarSig}, \text{PedestSig}) \equiv \\
&\quad \exists^2 \text{Tmp}. \text{ControlLogic}(\text{CarSig}, \text{Button}, \text{Tmp}, \text{end}) \wedge \\
&\quad \quad \forall^1 t. (\text{Tmp}(t) \leftrightarrow \text{CarSig}(s(t))) \wedge \\
&\quad \quad \forall^1 t. (\neg \text{CarSig}(t) \wedge \text{PedestSig}(t)) \vee \\
&\quad \quad \quad (\text{CarSig}(t) \wedge \neg \text{PedestSig}(t))
\end{aligned}$$

Fig. 4. The S1S translation of the traffic light example from section 2.1.

$$\begin{aligned}
&\llbracket (\text{name}, \text{Ins}, \text{Outs}, \text{Locals}, [\text{comp}_1, \dots, \text{comp}_n]) \rrbracket_{\text{WS1S}}^{\text{model}} = \\
&\quad \text{name}(\text{Ins}, \text{Outs}, \text{end}) \equiv \exists^2 \text{Locals}. \bigwedge_{i=1}^n \forall^1 t \leq \text{end}. \llbracket \text{comp}_i \rrbracket_{\text{WS1S}}^{\text{comp}}(t) \\
&\llbracket \text{Latch}(\text{In}, \text{Out}) \rrbracket_{\text{WS1S}}^{\text{comp}}(t) = t < \text{end} \rightarrow (\text{In}(t) \leftrightarrow \text{Out}(s(t))) \\
&\llbracket \text{Subckt}(\text{name}, \text{form_act}) \rrbracket_{\text{WS1S}}^{\text{comp}}(t) = \text{name}(\llbracket \text{form_act} \rrbracket, \text{end})
\end{aligned}$$

Fig. 5. The modifications of the semantics necessary for WS1S

for the outputs. For the translation of rows and literals, the names of the respective signals are additional parameters of the semantic functions. Latches, as mentioned previously, delay the input by one time unit. The initial value can be given by a reset table. Syntactically and semantically, reset tables are like combinational gates, except that they formalize a relation over just the initial time point.

3.2 Restriction to WS1S

The above translation models Core-BLIF circuit descriptions by modeling the evolution of the system state over infinitely many time points. Although this is a simple, appealing, semantics, the lack of tool support for S1S means we cannot directly use it for automated reasoning. In this section we show how the semantics can be recast in WS1S, whereby we can automate reasoning using the MONA system.

Modeling infinite behavior is not generally possible in WS1S since all sets are finite and hence any signal modeled must constantly take the value 0 after some time point. However, for verifying safety properties it is sufficient to model

all the finite prefixes of a circuit's behavior, which we can do by modeling its behavior from time 0 up to some point *end*, which is finite, but unbounded.

We do this as follows. We formalize *end* as a first-order variable in WS1S, given as an additional parameter to every predicate. As explained previously, components are modeled in S1S by constraints on the signals that have to be met at every time point. We now restrict this use of universal first-order quantification to time points up to *end* (so the values of the signal after *end* are not constrained). For latches we restrict the universal quantification to time points strictly before *end* since this constrains all successive time points up to *end* in the output signal. We summarize these modifications to the semantics in Figure 5 and give the WS1S translation of the traffic light example in Figure 6.

This restriction of the S1S semantics to finite interpretations is sensible. Under our translations, an infinite string is a model of the S1S semantics of a BLIF-MV circuit C iff all finite non-empty prefixes of the string are models of the WS1S semantics of C . We sketch the reasons for this below.

Proof Sketch. Observe that, since we do not constrain the time points after *end*, two finite interpretations of the signals of C are equivalent, if they are equal up to *end*. Hence, we introduce the following notation: For a formula ϕ , with free second-order variables $\bar{X} = (X_1, \dots, X_n)$ and a first-order variable *end*, we say $w \in (\{0, 1\}^n)^+$ models ϕ in the WS1S semantics, relative to *end*, written $w \models_{\text{WS1S}} \phi$, iff w encodes a model for ϕ , where X_i is interpreted as the i th track of w and *end* is interpreted as $|w| - 1$.

Now let $\text{pre}(w)$ be the finite non-empty prefixes of w . Formally we will show that $w \models_{\text{S1S}} \llbracket C \rrbracket_{\text{S1S}}$ iff for all $w' \in \text{pre}(w)$, $w' \models_{\text{WS1S}} \llbracket C \rrbracket_{\text{WS1S}}$.

To begin with, the (W)S1S translation of a circuit C can be rewritten into the form

$$\begin{aligned} \llbracket C \rrbracket_{\text{S1S}}(\bar{X}) &\equiv \exists^2 \bar{L}. \forall^1 t. \phi(\bar{X}, \bar{L}, t) \\ \llbracket C \rrbracket_{\text{WS1S}}(\bar{X}, \text{end}) &\equiv \exists^2 \bar{L}. \forall^1 t \leq \text{end}. \phi(\bar{X}, \bar{L}, t) , \end{aligned}$$

where \bar{L} represents the local signals of the overall circuit and ϕ is a quantifier-free formula that accesses only the signals at time points 0, $t - 1$, and t .

The left-to-right direction of the claim is straightforward. For the converse, assume we are given an infinite word w such that all non-empty finite prefixes satisfy $\llbracket C \rrbracket_{\text{WS1S}}$ relative to *end*. We have to show $w \models_{\text{S1S}} \llbracket C \rrbracket_{\text{S1S}}$. If there are no local signals, this is also straightforward by induction on the structure of the components. Otherwise, for every $w' \in \text{pre}(w)$ there is an instance of \bar{L} where ϕ is satisfied for all points up to the last position of w' . Let $N \subseteq (\{0, 1\}^{|\bar{L}|})^*$ be the set that contains all such instances for \bar{L} and, additionally, contains the empty word. Let E be the relation $(u, v) \in E$ iff $u \cdot x = v$ for some $x \in \{0, 1\}^{|\bar{L}|}$. (Figure 7 shows the graph for the example $\phi(\bar{L}, \text{end}) \equiv \forall^1 t. (t > 0 \wedge t \leq \text{end}) \rightarrow \neg L(t-1)$.)

From the fact that N is prefix-closed, it follows that the graph (N, E) is a tree (with the empty word as root). Moreover it is finitely branching (since the alphabet is finite) and for every depth there must be at least one node at this depth (since for every prefix of w there is a satisfying instance for \bar{L} of the same

$$\begin{aligned}
&\text{ControlLogic}(\text{PresentSig}, \text{Button}, \text{NextSig}, \text{end}) \equiv \\
&\quad \forall^1 t \leq \text{end}. (\text{PresentSig}(t) \wedge \text{Button}(t) \wedge \neg \text{NextSig}(t)) \vee \\
&\quad (\neg(\text{PresentSig}(t) \wedge \text{Button}(t)) \wedge \text{NextSig}(t)) \\
&\text{Lights}(\text{Button}, \text{CarSig}, \text{PedestSig}, \text{end}) \equiv \\
&\quad \exists^2 \text{Tmp}. \text{ControlLogic}(\text{CarSig}, \text{Button}, \text{Tmp}, \text{end}) \wedge \\
&\quad \forall^1 t < \text{end}. (\text{Tmp}(t) \leftrightarrow \text{CarSig}(s(t))) \wedge \\
&\quad \forall^1 t \leq \text{end}. (\neg \text{CarSig}(t) \wedge \text{PedestSig}(t)) \vee \\
&\quad (\text{CarSig}(t) \wedge \neg \text{PedestSig}(t))
\end{aligned}$$

Fig. 6. The WS1S translation of the traffic light example from Section 2.1.

length). From König’s lemma the tree must contain an infinite path, i.e. there must be an infinite string S , such that all strings on the path are finite prefixes of S . Thus S satisfies the constraints given by ϕ for all points up to an arbitrary bound end and, relying on the case without local signals, we can conclude that it does so for all points. Hence $w \models_{\text{S1S}} \llbracket C \rrbracket_{\text{S1S}}$. \square

Unfortunately, for certain kinds of circuit descriptions, the WS1S semantics allows more models than intended: if a string encodes a model under the WS1S semantics of the circuit up to end , it is not always the case that this is a prefix of a string under the S1S semantics. Consider the following example:

```
.names K L
0 0
0 1
.latch L K
```

This has exactly one model in the S1S semantics: K and L are constantly 0. However, in the WS1S semantics, for $\text{end} = 0$ (i.e., the string of length 1), we also have the model $K(0) = 0$ and $L(0) = 1$. The problem is that combinational tables define relations that need not be total on the input side, i.e. the gate can “refuse” certain inputs. Hence, a behavior that is consistent with the circuit description up to a given time point can later be “ruled out”. The graph of all models of this circuit, if we consider L as output, is the same as in Figure 7. The *undesired* models are those that are not on an infinitely long path and therefore cannot be extended arbitrarily far.

To eliminate these undesired models, we add to our specification the constraint that a finite behavior is only allowed if it can be extended beyond end up to an arbitrary time point new_end such that the extension is still a valid behavior of the circuit up to new_end . It turns out that this is easy to formalize in WS1S:

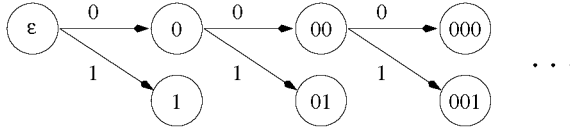


Fig. 7. Graph of finite models of the language $\phi(L, end) \equiv \forall^1 t. (t > 0 \wedge t \leq end) \rightarrow \neg L(t-1)$, including the empty word.

$$C'(\overline{X}, end) \equiv C(\overline{X}, end) \wedge \forall^1 new_end > end.$$

$$\exists^2 \overline{Y}. C(\overline{Y}, new_end) \wedge \forall^1 t \leq end. \overline{X}(t) \leftrightarrow \overline{Y}(t) .$$

Here C is the WS1S semantics (as defined above) of a circuit over the list of signals \overline{X} . The models of C' , relative to end , now have the property that they are precisely the prefixes of infinite behaviors.

As a consequence of the relation between the S1S and WS1S semantics of a circuit, we can check safety properties (as defined in [1]) with respect to the S1S semantics by checking them with respect to our WS1S translation: if MONA responds that no safety violation occurs in any finite prefix of the behavior of the circuit, then we can conclude the same for the infinite behavior.

4 Formal Analysis with MONA

Given a system description, we can use our B2M compiler to produce a set of formulae that express the semantics of the description's components in WS1S, in the syntax of MONA. We can then use MONA directly for simulation or for verification with respect to properties also expressed in WS1S. We now provide examples that illustrate the flexibility we gain by using a purely logical approach: by expressing appropriate constraints, we can restrict the set of possible circuit behaviors to the cases of interest; in this way there is a seamless transition from simulation to verification.

4.1 Simulation

As mentioned in Section 2.2, the models of a WS1S formula can be encoded by strings in a regular language. Given a formula, MONA computes a minimal deterministic finite automaton that accepts exactly the models of the formula and, from this automaton, MONA extracts minimal strings that are in, and outside, the language (if there are any). The strings in the language constitute simulated runs. The automaton generated constitutes a finite representation of all possible behaviors.

It is a simple matter to express, logically, constraints on the runs one is interested in. One can specify, for instance, runs of some particular length, or by

<i>Button</i>	0	0	0	0	0	x
<i>Car</i>	0	1	1	1	1	1
<i>Ped</i>	1	0	0	0	0	0
<i>end</i>	0	0	0	0	0	1

(a) Run of length 5

<i>Button</i>	0	1	0	1	0	1
<i>Car</i>	0	1	0	1	0	1
<i>Ped</i>	1	0	1	0	1	0
<i>end</i>	0	0	0	0	0	1

(b) Button pressed every other time unit

Fig. 8. Runs of the traffic light

expressing constraints on some of the input variables, and existentially quantifying over them, one can simulate outputs in response to certain inputs.

A simulation of the lights for, say, the time interval from 0 to 5 can be obtained using MONA by the formula $\text{Lights}(\text{Button}, \text{Car}, \text{Ped}, 5)$, which yields the run shown in Figure 8(a). Here x stands for an arbitrary Boolean value. This run corresponds to a situation in which the button is not pressed during the first five time units. If we desire, we can further restrict the circuit by providing more constraints on the input signals. For instance, to simulate the cases where the button is pressed every other time unit, we can specify

$$\begin{aligned} &\text{Lights}(\text{Button}, \text{Car}, \text{Ped}, \text{end}) \wedge \text{end} = 5 \\ &\wedge \forall t < \text{end}. \text{Button}(t) \leftrightarrow \neg \text{Button}(s(t)) \end{aligned}$$

In this case, MONA responds with the simulated run shown in Figure 8(b).

The fact that we can specify arbitrary WS1S constraints on both inputs and outputs and get a minimal automaton for the set of behaviors consistent with these constraints makes our approach to simulation quite flexible. For instance, if one has discovered some non-intended behavior, one can easily specify the property of the outputs that is violated and obtain the set of inputs that can cause this bug. Indeed, one can even stipulate the existence of some undesired behavior and generate a run for it.

4.2 Equivalence Checking

We can also use MONA to check equivalence of a given hardware description with some other sequential system.

To illustrate this, we have developed a slightly more sophisticated variant of the traffic light example, called **PhaseLight**: a new phase of the light, i.e. a time point where a light can change, is now controlled by an additional *timer*. When the pedestrians see red, the control logic stores, in an additional one bit register, whether the button was pressed at least once during this phase. We omit giving here the straightforward BLIF-MV description of **PhaseLight**.

We can now show, for example, that the simple traffic light circuit is a special case of the new circuit. Namely, if *Timer* is constantly 1, both circuits are equivalent.

$$\text{Lights}(\text{Button}, \text{Car}, \text{Ped}, \text{end}) \leftrightarrow (\exists^2 \text{Timer}. \forall^1 t \leq \text{end}. \text{Timer}(t) \wedge \text{PhaseLight}(\text{Timer}, \text{Button}, \text{Car}, \text{Ped}, \text{end}))$$

MONA can verify such formulae in negligible time and provides a counter-example (i.e. a string not in the language) in invalid cases.

4.3 Safety Properties

By reasoning about the finite traces of our systems, we can establish safety properties. For our light example, we can show, for example, that:

- (P1) The lights cannot simultaneous be green (or red) for the cars and pedestrians.
- (P2) If the cars' light is red, it turns green in the next phase.
- (P3) If the pedestrian's light is red and they press the button, then their light turns green in the next phase.

Note that (P2) and (P3) state eventualities, but since we stipulate when they must occur, they are indeed formalizable in WS1S.

The formalization of (P1)–(P3) is given in Figure 9: the lights are correct iff every assignment for the signals and *end* that constitutes a possible behavior of the *PhaseLight* circuit also satisfies the properties. For brevity, we define a predicate *NextPhase*, which states that, from time point *t* on, *t'* is the next rise of the timer signal plus one time unit. This unit delay is needed since there is a latch between inputs and outputs that delays the reaction of the control logic. (P2) for instance is formulated as follows: for arbitrary time points *t* and *t'* up to *end*, if the cars' light is red at time *t* and *t'* is the next phase after *t*, then the cars' light is green at *t'*. (Recall that red is encoded as 0 and green as 1 and $t' \leq \text{end}$ is contained in *NextPhase*). Again MONA verifies this automatically, requiring negligible time.

4.4 Performance

By using a general logic and a general purpose decision procedure we pay a performance price over more specialized algorithms for automata synthesis. However, for many examples of interest we get acceptable running times, which are typically around one order of magnitude slower then the running times of the VIS system. In Figure 10 we summarize the times for those circuits of the VIS example suite [13] that can be handled using our compiler and MONA without exceeding the physical memory⁵.

⁵ Running times are for a Ultra Sparc 2 450MHz workstation with 2,25 GB memory, typical memory usage for the examples was between 1 and 50 MB.

$$\begin{aligned}
\text{NextPhase}(\text{Timer}, t, t', \text{end}) &\equiv \\
&t < t' \wedge t' \leq \text{end} \wedge \text{Timer}(t' - 1) \wedge \\
&\forall^1 t''. ((t \leq t'' \wedge t'' < t' - 1) \rightarrow \neg \text{Timer}(t'')) \\
\text{LightsCorrect}(\text{Button}, \text{Timer}, \text{Car}, \text{Ped}, \text{end}) &\equiv \\
&\text{PhaseLight}(\text{Button}, \text{Timer}, \text{Car}, \text{Ped}, \text{end}) \rightarrow \\
&\forall^1 t \leq \text{end}. (((\text{Car}(t) \leftrightarrow \neg \text{Ped}(t)) \wedge \\
&(\forall^1 t'. ((\neg \text{Car}(t) \wedge \text{NextPhase}(\text{Timer}, t, t', \text{end})) \rightarrow \text{Car}(t')))) \wedge \\
&(\forall^1 t'. ((\neg \text{Ped}(t) \wedge \text{Button}(t) \wedge \text{NextPhase}(\text{Timer}, t, t', \text{end})) \rightarrow \text{Ped}(t')))))
\end{aligned}$$

Fig. 9. The formulation of the properties of the traffic light example in WS1S

Although run-times and example coverage are worse in our setting, we believe that substantial improvements are possible through compiler optimizations. Namely, the performance of MONA is quite sensitive to issues such as quantifier scoping and the way (equivalent) formulae are expressed. To gain some insights we compared the verification performance of compiler generated versus hand optimized MONA formulae. As an example, we verified the correctness of the sequential n -bit von Neumann adder for different values of n (by comparing results with a standard carry-chain adder). The hand optimizations included better quantifier scoping, constraint propagation and simplifications for combinational parts of the circuits. The times in Figure 11 show that there is considerable room for improvement, though there seems to be a general size frontier for the circuits that can be represented by MONA.

5 Conclusions

We have defined two formal semantics for BLIF-MV using the monadic logics S1S and WS1S. These provide precise, unambiguous interpretations over finite and infinite time intervals, and the WS1S semantics can directly be used for different kinds of symbolic analysis with the MONA system.

Our compiler provides a simple but flexible tool for understanding and experimenting with BLIF-MV specifications. However, the use of a simple high-level semantics and a general tool partially conflicts with the goal of optimal performance. As future work, we intend to investigate to what extent compiler optimizations, like those sketched in the previous section, can help bridge the performance gap.

Example	Description	Size	Property	Time
arbiter	Bus protocol	10	Mutual exclusion	1
counter	3 Bit	1	Approx. Liveness	< 1
crd	Crossroads	19	Self test const. 1 and Mutex	1
ctlp3	3 Philosophers	6	Reader unique	1
dcnew	Train-crossing	38	Safety1 (False) Safety2 (True)	38 32
8 Queens	Setting valid?	31	Exists valid setting	190
exampleS	req/ack-module	19	req until ack	4
mult 6x6	4 multipliers	each 4	First two equivalent	84
			Third buggy	85
			Fourth buggy	85
ping_pong	Simple game	6	Safety	< 1
ping_pong_new	... extension	7	Safety	< 1
tblOne_bug	Shows bug in the VIS system	1	Equivalence of two circuits	< 1
tlc	Traffic light controller by Conway & Mead	13	Safety/Eventualities	< 1

Fig. 10. Verification times (in seconds) of standard examples using MONA (size means size of BLIF input in KB).

n Bit	4	5	6	7	8	9	10	11
Compiler-generated	17	211	∞					
Hand optimized	<1	<1	1	6	22	74	239	∞

Fig. 11. Verification times for von Neumann adder; ∞ denotes exceeding memory resources.

References

- [1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 7 October 1985.
- [2] A. Ayari and D. Basin. Bounded model construction for monadic second-order logics. In *12th International Conference on Computer-Aided Verification (CAV'00)*, number 1855 in Lecture Notes in Computer Science, pages 99–113, Chicago, USA, July 2000. Springer-Verlag.
- [3] D. Basin and S. Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, pages 39–56. Research Studies Press/Wiley, 2000.
- [4] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3):255–288, 1998.
- [5] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.
- [6] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95, LNCS 1019*, 1996.
- [7] Y. Kukimoto. BLIF–MV. 1996.
Available at <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/>.
- [8] A. Meyer. Weak monadic second-order theory of one successor is not elementary-recursive. In *LOGCOLLOQ: Logic Colloquium*. LNM 453, Springer, 1975.
- [9] F. Morawietz and T. Cornell. On the recognizability of relations over a tree definable in a monadic second-order tree description language. Research Report SFB 340-Report 85, 1997.
- [10] A methodology for verification of real-time systems.
Available at <http://www-cad.eecs.berkeley.edu/Respep/Research/hsis/>.
- [11] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1967.
- [12] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4. MIT Press/Elsevier, 1990.
- [13] VIS Group. VIS user's manual.
Available at <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/>.
- [14] VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of CAV '96*, LNCS 1102, pages 428–432. Springer, 1996.

Checking Safety Properties Using Induction and a SAT-Solver

Mary Sheeran^{1,2}, Satnam Singh³, and Gunnar Stålmarck^{1,2}

¹ Prover Technology AB, Alströmergatan 22, 2tr, SE-112 47 Stockholm, Sweden
gunnar@prover.com

² Chalmers University of Technology, SE-412 96, Göteborg, Sweden
ms@prover.com

³ Xilinx Inc., 2100 Logic Drive, San Jose, California CA95124, USA
Satnam.Singh@xilinx.com

Abstract. We take a fresh look at the problem of how to check safety properties of finite state machines. We are particularly interested in checking safety properties with the help of a SAT-solver. We describe some novel induction-based methods, and show how they are related to more standard fixpoint algorithms for invariance checking. We also present preliminary experimental results in the verification of FPGA cores. This demonstrates the practicality of combining a SAT-solver with induction for safety property checking of hardware in a real design flow.

1 Introduction

We are interested in the problem of checking safety properties of large finite state machines using a SAT-solver. This has become an important research topic in recent years, and a number of apparently different approaches have been proposed [1, 2, 5, 7, 12]. Several of these methods seem promising, and experimental work to evaluate them is being carried out. We also need to develop a greater understanding of the problem and its various solutions in a more abstract sense. This paper contributes to this ongoing work in two ways. First, we explain some induction-based methods of safety property checking. Although applications of some of these methods have been reported [9], the methods themselves have not been properly documented in the literature. This paper attempts to remedy this. Second, we demonstrate the practicality of the approach by giving experimental results for the verification of real FPGA cores at Xilinx, Inc.

2 The Problem that We Would Like to Solve

Given a finite state machine M with initial states satisfying I and state transition relation T , we would like to check whether or not a property P holds for all reachable states. The transition relation T is a binary relation on the set of states S . We call a state that satisfies P a P -state, and a system in which all reachable states are also P -states is called P -safe. The reachable states are those that can be reached by T -transitions starting from an initial state.

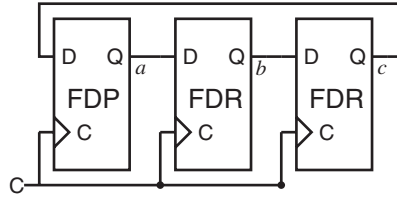


Fig. 1. A 3-bit ring counter

Example 1. Figure 1 shows a circuit for a 3-bit ring counter which has the property that only one bit is high at any given moment. When this circuit is reset into its initial state the Q output of the FDP flip-flop is set to 1 and the Q outputs of the two FDR flip-flops are set to 0. (We equate 0 with *False* and 1 with *True*.) This circuit has no inputs except for the clock C . Figure 2 shows a state transition diagram for this circuit which has one initial state $(1, 0, 0)$. In general, the state will be a finite vector of boolean variables. Transitions are

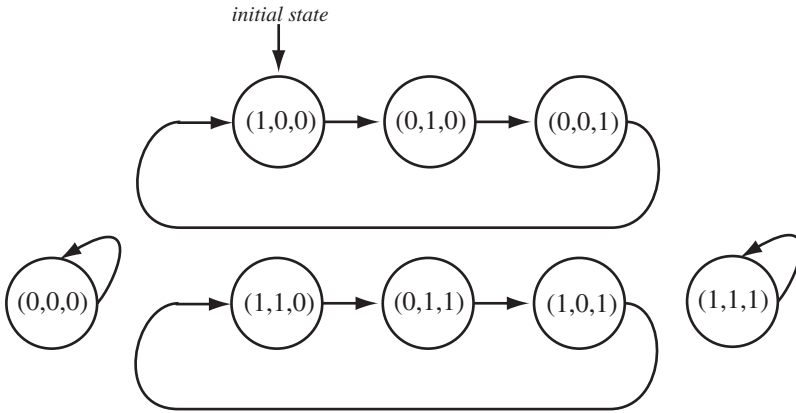


Fig. 2. State transition diagram for a 3-bit ring counter

shown as arrows between states. So, the circuit cycles between three reachable states. Let us call the three boolean state variables (a, b, c) as shown in Figure 1. Then, the property that only one bit should be high is represented by the formula $(a \oplus b \oplus c) \wedge \neg(a \wedge b \wedge c)$, informally “an odd-number of bits should be high, but not all three”, which we call *oneHigh*. (\oplus stands for exclusive or.) This property holds for all of the reachable states (on the top row of the diagram) and so the system shown is *oneHigh*-safe. An example of a property that does not hold for all reachable states is the formula $\neg c$. It holds for the initial state and for its successor, but not for the following state, so a suitable countermodel

to the assertion that $\neg c$ holds for all reachable states is the sequence of states $(1, 0, 0), (0, 1, 0), (0, 0, 1)$.

Generally such an error trace is a possible sequence of states starting at the initial state, in which all but the last state satisfy the required condition. When we check systems for P -safety, we would like to generate such a trace when the system turns out not to be P -safe. The question of how to get from a system description to a transition relation is not considered here. For an introduction to model checking in general, see reference [6].

2.1 Transition Relations and Paths

In order to be able to formulate the problem more precisely, we introduce notation for various types of paths through the graph of a transition relation. We write $T(x, y)$ to indicate that x is related to y by the transition relation T . Let us assume for notational convenience that the transition relation being examined is always T . For example the 3-bit ring counter presented in the previous section has a transition relation which relates the current state (a, b, c) to the next state (a', b', c') such that $a' = c$, $b' = a$ and $c' = b$. Now we define what it means for a sequence of states to be a path through T .

$$path(s_{[0..n]}) \triangleq \bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$$

Read \triangleq as “is defined to be”. $s_{[0..n]}$ is shorthand for the sequence of state (s_0, s_1, \dots, s_n) . We say that a path has length n if it makes n T -transitions. A path of length zero contains a single point and makes no transitions. To assert that a property Q holds of every point in a path, we write $all.Q(s_{[0..n]})$.

Later, we will have reason to restrict the paths in such a repeated composition to be *loop free*, so that every element of the path is distinct. We define

$$loopFree(s_{[0..n]}) \triangleq path(s_{[0..n]}) \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

The concatenation of two loop-free paths is not necessarily loop-free, but the sub-paths of a loop-free path are themselves loop-free. Thus

$$loopFree(s_{[0..(i+j)]}) \rightarrow loopFree(s_{[0..i]}) \wedge loopFree(s_{[i..(i+j)]}) \quad (1)$$

Sometimes we only want to talk about the ends of paths. So we are happy to view *path*, for instance, not only as a predicate on paths but also as a binary relation on points. We write $path_i(s_0, s_i)$ to indicate that there is a path from s_0 to s_i through i copies of T . This corresponds to quantifying away the internal points. So, for example

$$path_n(s_0, s_n) \leftrightarrow \exists s_1 \dots s_{n-1}. path(s_{[0..n]}) \quad (2)$$

Finally, we define what it means for a path to be a *shortest* path. In this case, we are only interested in the ends of paths. A path from a to b is shortest if it joins a and b and if a and b are not joined by any shorter path. Define

$$\text{shortest}(s_{[0..n]}) \triangleq \text{path}(s_{[0..n]}) \wedge \neg \left(\bigvee_{0 \leq i < n} \text{path}_i(s_0, s_n) \right) \quad (3)$$

Shortest paths are also loop-free. Note that the definition of *shortest* in fact contains many existential quantifiers, because we have repeatedly used $\text{path}_i(s_0, s_n)$. For a finite transition relation T , there exists a largest k for which $\text{shortest}(s_{[0..k]})$ holds for some sequence of states. In other words, there is a longest shortest path in a finite state transition graph. The length of that path is usually called the *diameter* of the graph. The diameter of the state transition graph shown in figure 2 is 2.

2.2 Formulating the Problem

Let T be a transition relation on the set of states S . We assume that the domain of T is the entire set of states S , so that every state has a successor through T . Let I characterise the initial states, and P the property of states that we want to check.

We want to show that starting in an initial state and repeatedly applying the transition relation always leads to a state satisfying P . That is, we want to prove

$$\forall i. \forall s_0 \dots s_i. (I(s_0) \wedge \text{path}(s_{[0..i]}) \rightarrow P(s_i))$$

where $i \geq 0$ and the s_i range over states. Or we can work backwards from the bad states. We want to show that starting in a state violating P and working backwards through T always leads to a non-initial state, that is

$$\forall i. \forall s_0 \dots s_i. (\neg I(s_0) \leftarrow \text{path}(s_{[0..i]}) \wedge \neg P(s_i))$$

Both of these turn out to be the same thing as proving

$$\forall i. \forall s_0 \dots s_i. \neg(I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i))$$

This gives a more symmetrical view of the problem. In words, we want to show that there are no paths that start in an initial state and end in a non- P -state.

3 A First Solution

How can we divide our problem up into smaller sub-problems?

A possible first solution is to check that

$$\forall s_0 \dots s_i. \neg(I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)) \quad (4)$$

holds for $i = 0$, $i = 1$, $i = 2$, and so on. This corresponds to checking that $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ is contradictory (or that $\neg(I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i))$ is a tautology) for each i , for arbitrary s_0 to s_i . If the property is violated somewhere in the reachable states, we will eventually find an i for which $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ is satisfiable. Then, we know that there is a path of length i from an initial state to one violating P , and indeed the assignment of values to $s_{[0..i]}$ that makes the formula satisfiable is such a path, and can be used for debugging purposes. Also, we know that there is no shorter such error-trace. For the special case of simple safety properties, Bounded Model Checking, a particular form of model checking based on SAT-solving proposed by Clarke and his collaborators [2], reduces to a similar kind of iteration and satisfiability check.

If the system is P -safe, formula (4) will always hold. The question is how do we know when it is safe to stop incrementing i and conclude that the system is P -safe? It is no good waiting for $I(s_0) \wedge \text{path}(s_{[0..i]})$ to be contradictory, say. Given that there is an initial state, this will never happen, as we assume that every state has a successor through T , so there are always loops in both the reachable and the unreachable state space.

A better strategy is to stop when $I(s_0) \wedge \text{loopFree}(s_{[0..i]})$ becomes contradictory. Then, we stop when we have checked every loop-free path (and thus every state) in the reachable states. We can then safely conclude that the system is P -safe. Similarly, we can keep checking until $\text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$ becomes contradictory, and stop, again with a positive answer, when we have checked all states reachable backwards from those violating P . This solution is given in pseudo-code below (Algorithm 1). The function `Sat` corresponds to a call to a SAT-solver. The function `Sat` takes an expression and returns `True` if there exists an assignment to the variables (in this case $s_{[0..i]}$) which make the whole expression true. Here, the trace $c_{[0..i]}$ is an assignment to the variables $s_{[0..i]}$ that

Algorithm 1 First algorithm to check if system is P -safe

```

i:=0
while True do
  if not Sat( $I(s_0) \wedge \text{loopFree}(s_{[0..i]})$ ) or not Sat( $(\text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i))$ ) then
    return True
  end if
  if Sat( $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ ) then
    return Trace  $c_{[0..i]}$ 
  end if
   $i = i + 1$ 
end while

```

makes $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ true, and so is a suitable error trace.

Let us consider the case when the answer is `True`. We prove that for all i

$$I(s_0) \wedge \text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. That is, we show that there is no loop-free path, starting from the initial state, in which the final state violates P . If there is no such path, then the system must be P -safe, and thus the method is sound. When the answer is True, we know from the condition in the first if statement that for some smallest k one of $I(s_0) \wedge \text{loopFree}(s_{[0..k]})$ and $\text{loopFree}(s_{[0..k]}) \wedge \neg P(s_k)$ must be contradictory (or not satisfiable). We also know, from the second if statement, that for $i < k$ it is the case that

$$I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. A consequence is that for $i < k$

$$I(s_0) \wedge \text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i) \quad (5)$$

is also contradictory. If there are no paths linking an initial state to a non- P -state, then there are no loop-free paths doing so either.

It remains to be shown that equation (5) holds for $i \geq k$. For $i \geq k$, we know (by equations 1 and 2 and some quantifier manipulation) that

$$\begin{aligned} I(s_0) \wedge \text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i) \\ \rightarrow I(s_0) \wedge \text{loopFree}(s_{[0..k]}) \wedge \text{loopFree}(s_{[k..i]}) \wedge \neg P(s_i) \end{aligned}$$

and that

$$\begin{aligned} I(s_0) \wedge \text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i) \\ \rightarrow I(s_0) \wedge \text{loopFree}(s_{[0..m]}) \wedge \text{loopFree}(s_{[m..i]}) \wedge \neg P(s_i) \end{aligned}$$

for some m . But at least one of those right hand sides is unsatisfiable since one of $I(s_0) \wedge \text{loopFree}(s_{[0..k]})$ and $\text{loopFree}(s_{[0..k]}) \wedge \neg P(s_k)$ is. We conclude that for all i

$$I(s_0) \wedge \text{loopFree}(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. Thus a True answer does indeed indicate that the system is P -safe. The method is also complete. It returns True if the system is P -safe, and an error trace if not. The restriction to loop-free paths is necessary for completeness.

This algorithm is in a sense bidirectional; it can be thought of as working both forwards from the initial state and backwards from the bad states at the same time. Indeed, it is pleasingly symmetrical. We could swap the initial and the bad states, and replace the transition relation by its converse, and still get the same algorithm.

In the ring counter shown in figure 2, using this algorithm to check that *oneHigh* holds of all reachable states returns True when i becomes 3 since 2 is the length of the longest loop-free path starting from the initial state, and of the longest loop-free path ending in a non-*oneHigh* state. Checking for the formula $\neg c$ that we considered earlier returns the sequence of states $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ as the assignment $s_0 = (1, 0, 0), s_1 = (0, 1, 0), s_2 = (0, 0, 1)$ is a satisfying assignment for the formula $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ in this case.

4 Improving on this Solution

How can we improve this algorithm, bearing in mind that we will use a SAT-solver to check the formulas? Well, we can make the two termination conditions a bit tighter.

Let us think operationally for a moment, and imagine traversing the state transition graph. In the forward direction, we don't want to go back into an initial state as we would then be considering a longer path than necessary. We could in that case consider only the end part of the path starting from the second point that is an initial state. The original termination condition was $I(s_0) \wedge \text{loopFree}(s_{[0..i]})$ and now we want to replace it by

$$I(s_0) \wedge \text{all}.\neg I(s_{[1..i]}) \wedge \text{loopFree}(s_{[0..i]})$$

(In the special case where there is only one initial state, then this change is unnecessary, as the restriction to proper paths prevents us from returning to the initial state.) Similarly, in the backwards direction, we are uninterested in paths that have a non- P -state somewhere in the middle. We only want to consider paths in which all but the last state satisfy P . The new termination condition is then

$$\text{loopFree}(s_{[0..i]}) \wedge \text{all}.P(s_{[0..(i-1)]}) \wedge \neg P(s_i)$$

The resulting algorithm is given as Algorithm 2.

Algorithm 2 An improved algorithm to check if system is P -safe

```

i:=0
while True do
  if not Sat( $I(s_0) \wedge \text{all}.\neg I(s_{[1..i]}) \wedge \text{loopFree}(s_{[0..i]})$ )
  or not Sat( $(\text{loopFree}(s_{[0..i]}) \wedge \text{all}.P(s_{[0..(i-1)]}) \wedge \neg P(s_i))$ ) then
    return True
  end if
  if Sat( $I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg P(s_i)$ ) then
    return Trace  $c_{[0..i]}$ 
  end if
  i = i + 1
end while

```

One of us was sorely tempted to make use of facts proved in earlier iterations to make further restrictions in both termination conditions, restoring a pleasing symmetry. But this turns out to be a bad idea in practice because of the need to rely on previous iterations. When a circuit requires a very high induction depth to prove a property, it is simply too expensive to iterate all the way up to that depth, from zero. The proofs that find that we cannot yet terminate are much slower than the successful proofs of termination conditions. So, we should instead concentrate on removing the need to iterate upwards from zero depth!

We need to change the check for bad paths so that it can find bad paths of length 0 up to i , and not just of length exactly i . It turns out to be convenient to switch the order of the check for bad paths and the check for termination.

Algorithm 3 An algorithm that need not iterate from 0

```

i = some constant which can be greater than zero
while True do
  if  $\text{Sat}(I(s_0) \wedge \text{path}(s_{[0..i]}) \wedge \neg \text{all}.P(s_{[0..i]}))$  then
    return Trace  $c_{[0..i]}$ 
  end if
  if not  $\text{Sat}(I(s_0) \wedge \text{all}.\neg I(s_{[1..(i+1)]}) \wedge \text{loopFree}(s_{[0..(i+1)]}))$ 
  or not  $\text{Sat}((\text{loopFree}(s_{[0..(i+1)]}) \wedge \text{all}.P(s_{[0..i]}) \wedge \neg P(s_{i+1}))$  then
    return True
  end if
   $i = i + 1$ 
end while
    
```

Now, we are no longer obliged to iterate all the way up from zero, as we have removed the dependence between iterations. The length of the longest serial connection of latches (or other delay elements) is usually a lower bound on the number of iterations needed, so that is a good starting point. The algorithm is still sound, even if we start at “too high” a value of i . In that case, though, the error trace returned is no longer guaranteed to be of minimal length. Each iteration also begins to look more like an inductive proof. Rewriting some of the subproblems to equivalent ones gives Algorithm 4. Here the call to Taut invokes a SAT-solver to establish whether its argument expression is always true.

Algorithm 4 A forwards version of the algorithm

```

i = some constant which can be greater than zero
while True do
  if  $\text{Sat}(\neg(I(s_0) \wedge \text{path}(s_{[0..i]}) \rightarrow \text{all}.P(s_{[0..i]})))$  then
    return Trace  $c_{[0..i]}$ 
  end if
  if  $\text{Taut}(\neg I(s_0) \leftarrow \text{all}.\neg I(s_{[1..(i+1)]}) \wedge \text{loopFree}(s_{[0..(i+1)]}))$ 
  or  $\text{Taut}((\text{loopFree}(s_{[0..(i+1)]}) \wedge \text{all}.P(s_{[0..i]}) \rightarrow P(s_{i+1}))$  then
    return True
  end if
   $i = i + 1$ 
end while
    
```

Now we can begin to see the inductive shape of the proof. The first if statement is the base case. It checks that P holds in the first $i + 1$ states. The second disjunct in the condition in the next if statement checks that after $i + 1$ P -states in a row one is guaranteed to reach another P -state. By induction, we conclude

that every loop-free path starting at the initial state contains only P -states. We call this *strengthened induction with depth i* , and it proves that the system is P -safe. The word *strengthened* refers to the restriction to loop-free paths, which is an additional constraint on the unrollings of the transition relation. Without this constraint, induction with depth gives an incomplete method. Later, we shall see a stronger variant of induction. However, even this the weakest form of induction works well for some kinds of hardware verification, and our experimental results from core verification use only this form of induction so far. We don't yet know whether or not the stronger forms of induction are useful in practice. Returning to the remaining disjunct in the second if statement, it checks whether or not we can stop because all of the loop-free paths in the reachable states have already been checked (by the base cases). This view of the algorithm has more of a left-to-right character than the original one, but the two algorithms are in fact the same! Indeed, there is also a right-to-left version, which you can get by replacing the condition in the first if statement by $all.\neg I(s_{[0..i]}) \leftarrow path(s_{[0..i]}) \wedge \neg P(s_i)$. Now, view this as the base case of the induction, and the first disjunct of the second if statement as the step. In fact, though, both versions behave identically as the condition that we have just introduced traps exactly the same bad paths as the previous one. It is in the left-to-right form that we have presented the algorithm earlier[12]. We feel that the more symmetrical presentation given here is enlightening. In particular, it has helped us to understand the importance of the double termination check which gives us an algorithm that can be seen as working both forwards and backwards.

Returning to the ring counter example and again checking the *oneHigh* property, we find that the tighter termination condition now allows us to terminate when i is 0. This is because there are no paths of length 1 connecting a *oneHigh* state to a non-*oneHigh* state. If the property being checked happens to coincide exactly with the reachable states, as in this example, then there are no paths from a P -state to a non- P state, so induction with depth 0, which is just ordinary induction, will succeed. Restricting the backwards termination condition to consider only paths in which all but the last state satisfy P is an important refinement of the algorithm.

The limiting factor for this algorithm is the cost of adding the extra constraints in the termination conditions that constrain the paths to be loop-free. This needs in the order of n^2 inequalities between states for path length n . Since Stålmarck's method copes well with very large formulas, the limit is not as constraining as it might appear [11]. Somewhat surprisingly, most of the examples that we have tried so far have needed relatively low induction depths. However, there will clearly be deep systems that we simply cannot cope with.

The algorithms that we have presented so far are suited for use with a SAT-solver. The formulas that need to be checked are given by the definitions of *path*, *loopFree* and *all.P*.

At Prover Technology AB, the algorithm is implemented in the prototype LUCIFER tool for checking safety properties of Lustre programs [10]. LUCIFER has been used in a number of industrial verification projects at Prover Technology

AB. It is currently being evaluated at a large aerospace company, for use in the verification of control programs and a product incorporating these algorithms and others is being developed. LUCIFER is also in use in a project to develop a design flow incorporating formal verification for FPGA cores at Xilinx, Inc. [9]. Later, in section 7, we present experimental results from this project.

5 A Second Stronger Solution

The algorithm that we have just presented has the advantage that it can be implemented using a plain SAT-solver. However, it is unsatisfactory when one considers the necessary number of iterations before it terminates, for a P -safe system. The number of iterations required is either the length of the longest loop-free path starting from an initial state (and proceeding through non-initial states), or the length of the longest loop-free path consisting of all P states followed by a non- P -state, whichever is the shorter. But this could easily be far too many iterations! The longest loop-free path between a pair of states may be *much* longer than the shortest path between them, so the algorithm may needlessly consider long paths. We would like to consider only *shortest* paths between pairs of states. This insight leads to a new solution. The adaption of the algorithm to consider only shortest paths is straightforward. Everywhere we had *loopFree*, we substitute *shortest*. The reasoning is just as before, except that

Algorithm 5 A version of the algorithm that considers shortest paths

```

i = some constant which can be greater than zero
while True do
    if Sat( $I(s_0) \wedge path(s_{[0..i]}) \wedge \neg all.P(s_{[0..i]})$ ) then
        return Trace  $c_{[0..i]}$ 
    end if
    if not Sat( $I(s_0) \wedge all.\neg I(s_{[1..(i+1)]}) \wedge shortest(s_{[0..(i+1)]})$ )
    or not Sat( $(shortest(s_{[0..(i+1)]}) \wedge all.P(s_{[0..i]}) \wedge \neg P(s_{i+1}))$ ) then
        return True
    end if
     $i = i + 1$ 
end while

```

this time we prove that for all i

$$I(s_0) \wedge shortest(s_{[0..i]}) \wedge \neg P(s_i)$$

is contradictory. Considering only shortest paths does not reduce the set of states considered, so this condition still captures P -safety. The big difference is that for a P -safe system, this algorithm may terminate much earlier.

Note, however, that using *shortest* in the algorithm means that we have introduced many existential quantifiers. The algorithm is no longer suitable for

use with a plain SAT-solver, but needs quantifier elimination. Work on the FixIt tool shows that one can get quite far in SAT-based reachability analysis using relatively simple quantifier elimination [1]. Another alternative is to use the QBF-specific part of Stålmarck's translation of finite domain many sorted first order logic into propositional logic [13]. We will not consider either option further in this paper, but note that much experimental work remains to be done.

We call the *forward diameter* of a graph the length of the longest shortest path starting in an initial state and proceeding through non-initial states, and the *backward diameter* the length of the longest shortest path consisting of all P -states followed by a non- P -state. The number of iterations needed for a P -safe system is the minimum of the forward and backward diameters.

Algorithm 5 still considers entire paths. We can think of modifying it so that it instead only considers the two end points of a path. This makes sense as we are now considering shortest paths. Having made that step, we can make one last quantifier elimination in each of the termination conditions. We would rather not be forced to iterate very far by a system that has a very long shortest path from an initial state to x , if x is reachable much earlier from a *different* initial state. We would somehow like to bundle all the initial states together. Our definition of *shortest* gives us shortest paths from a single state. Now we would like to consider shortest paths from a set of states. We modify the definition of *shortest*. S characterises the set of states.

$$shortest'_n(S, b) \triangleq \exists a. (S(a) \wedge path_n(a, b)) \wedge \neg \exists a. (S(a) \wedge \bigvee_{0 \leq i < n} path_i(a, b))$$

Now, $shortest'_n(S, b)$ also characterises a set of states. The important point to note is that we have added two further existential quantifiers. Similarly, we will write $shortest'_n(a, S)$ for the assertion that there is a shortest path from state a into the set characterised by S . For simplicity, we also omit the constraints on internal points on paths. The result is shown as Algorithm 6.

Algorithm 6 A set-based version of the algorithm that considers shortest paths

```

i = some constant which can be greater than zero
while True do
  if Sat( $I(s_0) \wedge path(s_{[0..i]}) \wedge \neg all.P(s_{[0..i]})$ ) then
    return Trace  $c_{[0..i]}$ 
  end if
  if not Sat( $shortest'_{i+1}(I, s_{i+1})$ )
  or not Sat( $(shortest'_{i+1}(s_0, \neg P))$ ) then
    return True
  end if
   $i = i + 1$ 
end while

```

This algorithm is similar to the standard method of checking safety properties using fixpointing and a simultaneous forward and backward analysis. The

existential quantifiers that we have just added in the termination conditions have moved us from a path-based algorithm to one that operates on sets of states.

So, we have seen two kinds of solution, one that works directly with quantifier free formulas but that may need to iterate too far in practice in some cases, and a much stronger solution that seems to demand quantifier elimination. In between these two extremes, there is a range of solutions that can be explored. We want to look at various versions of constrained iteration of relations between *loopFree* and *shortest*.

One way to think about this range is to consider that the constraints that we apply are of the form “paths of length j or greater do not have any paths that are shorter than j between their end points”. Now, choosing j to be 1 gives us the constraint that we used in our purely SAT-based solutions. We constrain all paths of length 1 or greater to have unequal end points (so that they obey $\neg path_j(x, x)$). If on the other hand, we choose j to be the length n of the path that we are considering, then we get the strongest form of induction. We demand that the entire path be a shortest path, so that its end points are not joined by any paths of length less than n . In between these two extremes, we can choose different values of j , to give a range of constraints. We have seen that setting j to be one gives loop-free paths. Choosing $j = 2$ gives what we call locally shortest paths; the consequence is that all paths of length 2 or greater obey the negation of the transition relation T and the entire path has unequal end points. This again is a constraint that can be expressed in pure propositional logic, and it is strictly stronger than the *loopFree* constraint. All of these constrained forms of iteration can be plugged into our basic algorithm. As yet, our experiments have been restricted to induction strengthened by the restriction to loop-free paths.

6 Related Work

Deharbe and Moreira have suggested using induction (with depth one) to check invariant properties of transition systems [8]. They modify a standard model checking algorithm to use induction for properties of the form $AG\ p$ (informally, “ p is globally true along all paths”). Thus, this work is done in a context in which sets of states and image computations are expressed using BDDs. The authors point out that their method is incomplete and do not, to our knowledge, consider additional path constraints as we do.

In Bounded Model Checking (BMC), the user specifies a number of time steps, k , for searching from initial states for countermodels to properties [2, 3]. This work has alerted many to the possibilities of SAT-solvers in model checking. Reference [4] concentrates particularly on safety properties. It applies the method to the checking of safety properties of a PowerPC microprocessor at Motorola. The method used is either to search for a finite-length counter-example, exactly as we do, or to prove that the property is an inductive invariant, using induction with depth zero, that is ordinary induction. Induction with depth seems not to be considered. The authors point out that the technique is not complete. We know, however, that the authors considered restrictions to loop-free paths.

When it comes to termination conditions, the original BMC paper proposes to use the graph diameter as the length of the longest necessary unrolling. This would correspond, in our formulation, to terminating when $\text{shortest}(s_{[0..i]})$ becomes unsatisfiable. Our termination conditions are rather more refined in that they take account of the initial states and of the property to restrict attention only to more relevant sub-graphs. The restrictions to paths of the form “I then all not I” or “all P then not P” turn out to be important in practice, though we must admit that we have tried them mostly in the context of the weaker restrictions to loop-free paths.

For an interesting future research direction on the theme of variations on induction and SAT-solving, the reader is referred to recent work by Bjesse and Claessen (in this volume) on improving induction using a method first proposed by van Eijk [5].

Many of the methods that we propose here are already in use in automatic test pattern generation (ATPG) and one of our next steps will be to study SAT-based ATPG.

7 Results from FPGA Core Verification

At Xilinx the LUCIFER tool has been used to help verify the correctness of FPGA circuit cores – intellectual property that Xilinx distributes or sells. It is particularly important to ensure the correctness of these cores since users expect intellectual property to have undergone rigorous testing.

In a recent project many of the basic building block circuits, called BaseBLOXs, were verified with LUCIFER. Typical components include bus multiplexors, adders, subtractors, accumulators, comparators, complementors and counters. Full details of the BaseBLOX components are available from Xilinx [14].

Although in principle these circuits may not seem terribly challenging for formal verification, industrial versions include many extra inputs and configuration information which makes these circuits harder to verify. For example, the counter could have its count direction changed dynamically, it could also have a new value loaded dynamically on any clock tick, the amount to increment the count by can also be dynamically altered and two outputs indicate when certain count thresholds have been reached. Furthermore, the counter can have a clock enable as well as synchronous or asynchronous clears and sets to dynamically determined values. These factors inflate the number of primary inputs and outputs and increase the number of state elements.

The LUCIFER system accepts input in the Lustre language and generates a proof log which contains a countermodel if one exists plus other statistics about the verification. A system called Argus [9] was developed (see Figure 3) which translates Xilinx circuit designs in the EDIF netlist format into behaviourally equivalent Lustre. The Argus system then poses a question to the LUCIFER system to perform equivalence checking for the two input designs. When a counter model is found it is read from the generated proof log and translated into a simulation script for the ModelSim VHDL/Verilog simulator.

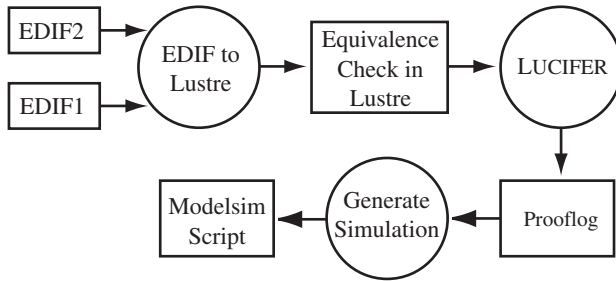


Fig. 3. Architecture of the Argus EDIF netlist equivalence checker

The core verification flow at Xilinx is shown in Figure 4. The Core Generator system is called upon to generate a specific instance of some core e.g. by specifying the size of a multiplier or the degree of pipelining required. The outputs of the Core Generator are (i) a highly optimised EDIF circuit netlist suitable for implementation on an FPGA and (ii) a VHDL behavioural description which should faithfully model the behaviour of the circuit contained in the EDIF netlist. This may easily not be the case since the behavioural descriptions are developed entirely independently of the optimised implementation circuits. The verification system that we have put in place performs equivalence checking to ensure that for specific instances of cores the implementation netlists are correctly modeled by the behavioural descriptions.

The flow involves taking a behavioural VHDL specification of a core's behaviour and synthesising a particular instance of it to produce an EDIF netlist. This acts as the specification of the required behaviour. Then the actual highly optimised structural VHDL (or other language) code for the corresponding core implementation is also elaborated into an EDIF file. The Argus equivalence checker then processes these two EDIF files. If the system finds a counter-model, then a simulation script identifying the sequence of events that led to the discrepancy is produced. This script can be used by the core designer or verifier from the ModelSim VHDL simulator to help identify the source of the problem.

A key aspect of our flow is that the verification engineer or core developer does not need to learn about any formal logic since the safety property that we wish to check (that the two circuits under examination always have the same output) is automatically generated by our system. Furthermore, when a countermodel is found the results are presented by running a familiar VHDL simulator. This makes the system more accessible to engineers. A weakness of the system is that manual intervention is sometimes required when the generated behavioural VHDL is not synthesisable, although this has rarely been the case in practice.

To illustrate the performance of LUCIFER we present the results of using it as a part of the Argus system to perform equivalence checking of various cores against their behavioural descriptions. Some of the results are shown in Table 1. These experiments were run on a dual processor Ultra-60 SparcStation with 2

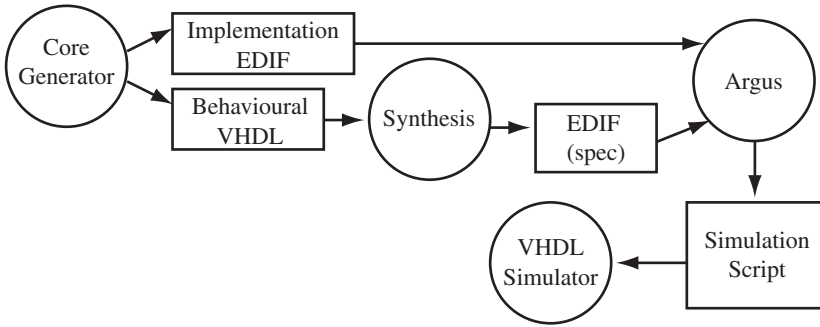


Fig. 4. Core Verification Flow at Xilinx

GB of RAM. Note that we verify *fixed size instances* of circuits, as you must expect of a system based on propositional logic.

Table 1. Experimental Results

Experiment	Verification Time (seconds)	Induction depth
64-bit up counter	1.03	0
64-bit up-down counter	75.39	0
11-bit loadable counter	2938	0
11-bit increment by 14 counter	0.16	0
11-bit increment by 14 loadable counter	2965	0
11-bit pipelined counter (1 stage)	46.21	2
11-bit pipelined counter (2 stages)	283	6
11-bit pipelined counter (3 stages)	526	10
1000-bit Johnson counter	44.45	2

The verification times for various n -bit up counters (not loadable, count by 1 only, no threshold outputs) are quite favorable. The netlist produced by the core generator system and behavioural synthesis system were examined. Both systems produced designs that contained D-type flip-flop state elements, but the core generator system used a D-type flip-flop with an enable signal whereas the synthesised version produced a D-type flip-flop active on a negative edge. These components are encoded with different logical representations so the system has to perform some non-trivial checking to ensure that the two counters are the same.

A counter which can be made to count up or down depending on a control signal has a more complicated state space, as suggested by the timing results. However, even for a 64-bit up-down counter, the verification takes little time (around one minute).

Keeping the count direction fixed but allowing the counter to be loaded with a new value on any clock tick proves to be more challenging for LUCIFER. It took about 50 minutes to verify an 11-bit loadable counter. Both the core generator and behavioural versions produce the same number of state elements but the core generator uses FDE flip-flops and the synthesiser used FD (D-type) flip-flops.

Verifying counters with specific increment values posed no problem for the verification with a 11-bit increment by 14 count taking just 0.16 seconds. The next experiment combines the up-down feature of the counter with the ability to dynamically load the counter. Once again the dynamic load requirement pushes the verification time to 2,965 seconds.

Next, we introduce a pipeline stage at the end of the counter. The verification is posed in such a way that the first output of the two circuits is ignored but then every subsequent output is required to be the same. Adding a pipeline stage causes a marked increase in verification effort. The time needed to verify a non-pipelined 11-bit adder is 0.17 seconds but adding one pipeline stage causes the verification time to shoot to 46.21 seconds. LUCIFER uses an induction depth of 2 to verify these counters. Adding another stage takes the verification time up to 283 seconds. We believe that this is due to the particular way the pipelining property has been expressed to LUCIFER and this is something we expect to be able to improve upon dramatically in the next iteration of verifications.

To study a larger example with unreachable states we verified Johnson counters at a variety of sizes (also known as twisted-ring counters). A Johnson counter counts in a sequence in which only one bit changes per clock tick. An n -bit Johnson counter cycles through $2n$ states, giving $2^{(n/2)} - n$ unreachable states. A 1,000 bit Johnson counter took 44.45 seconds to verify with induction depth 2. Verification of the ring counter presented earlier yields similarly encouraging results.

8 Discussion and Conclusion

We first presented a method of safety property checking based on induction with depth, strengthened with a constraint that all states in a path be unique. This method is complete. It is the method implemented in the prototype LUCIFER tool for analysis of Lustre programs [10]. The work on FPGA core verification described above is based on LUCIFER. Thus, these first results make use of strengthened induction with depth, and show that it can cope with non-trivial equivalence checking. The results also demonstrate that the method can be incorporated into a real design flow. The fact that erroneous behaviour found during attempted verification can be analysed in a standard VHDL simulator is particularly important. Our effort has so far been concentrated on the considerable task of building the infrastructure to automatically verify real cores in an existing design environment. The results reported have been produced only just in time for inclusion here, and we have not yet had time to analyse them. It is intended to continue this work by verifying a sequence of increasingly complicated cores. Those that are next on the list are large shifters and state machine based

controllers. Thus, the development of both the methods and the infrastructure will be driven by real case studies. We expect to have to trim the safety property checking methods to match exactly this application to core verification. By doing so, we expect to reduce the verification times reported here considerably. We have a great deal of experimental work ahead, both in applying our methods and in comparing with more standard BDD-based verification methods.

We have also shown that the strongest form of induction is very close to a standard backwards and forwards analysis using fixpoints. The FixIt system, developed by Bjesse and Eén, implements many safety property checking algorithms, including BMC, strengthened induction with depth, and SAT-based versions of standard fixpointing algorithms. It makes use of a relatively simple quantifier eliminator, but still gives very promising results [1]. FixIt is now being used as a basis for experiments in SAT-based verification. We plan to use FixIt to investigate a variety of induction-based methods, including those presented in the previous section.

When using these induction-based methods, one can vary not only the induction strength, but also the transition relation, T . We can try to make the relation smaller (in ways that do not affect the final result of the analysis) so as to prune away paths, and so possibly reduce the necessary induction depth. The application of van Eijk's method by Bjesse and Claessen can be seen as an example of this [5]. Also, making the transition relation (viewed as a set of pairs of states) larger, while leaving the transitive closure unchanged, may cause the depth of induction needed to prove a property to be reduced. So, the challenge is to increase the size of the relation while leaving its transitive closure unchanged. It seems likely that we can use insights from relational algebra here. One can also think of eliminating some of the quantifiers in the unwindings of the transition relation (as distinct from in the constraints). This would give a sort of hybrid between the usual fixpoint methods and the inductive methods presented here. And there are many tricks from the world of BDD-based model checking that we haven't even considered yet! We have begun to think that a possible way to proceed might be to go back to basics and think of all these methods, and possible new ones, in terms of propositional temporal logic theorem proving. Insights from proof theory might then give us a new way to compare the different methods.

Acknowledgments

This research has been funded by Prover Technology AB, Xilinx, Inc., Chalmers University of Technology, the Swedish funding agency TFR, and the EU LTR project SYRF (Synchronous Reactive Formalisms). Carl Johan Lillieroth built major parts of the infrastructure for core verification. Many thanks to Per Bjesse, Koen Claessen and Gordon Pace for their constructive criticism of an earlier draft. Thanks also to Nicolas Halbwachs, Pascal Raymond and Bernie New for enlightening discussions.

References

1. P. A. Abdulla, P. Bjesse and N. Eén: Symbolic Reachability Analysis based on SAT solvers, In Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'00, LNCS, Springer-Verlag, 2000.
2. A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu: Symbolic Model Checking without BDDs. In Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99, number 1579, LNCS, Springer-Verlag, 1999.
3. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu: Symbolic model checking using sat procedures instead of BDDs. Design Automation Conference, DAC'99, IEEE Press, 1999.
4. A. Biere, E.M. Clarke, R. Raimi and Y. Zhu: Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In Proc. Int. Conf. on Computer-Aided Verification, CAV'99, LNCS, Springer-Verlag, 1999.
5. P. Bjesse, K. Claessen: SAT-based Verification without State Space Traversal. In Proc. Int. Conf. on Formal Methods in Computer Aided Design of Electronic Circuits, FMCAD'00, LNCS, Springer-Verlag, 2000.
6. E. Clarke, O. Grumberg and D. Peled: *Model Checking*, MIT Press, 1999.
7. W.J. Fokkink and P.R. Hollingshead: Verification of Interlockings: From Control Tables to Ladder Logic Diagrams, in (J.F. Groote, S.P. Luttik and J.J. van Wamel, eds) Proc. 3rd Workshop on Formal Methods for Industrial Critical Systems, FMICS'98, Amsterdam, 1998.
8. D. Deharbe and A. Martins Moreira: Using Induction and BDDs to Model Check Invariants, In H. Li and D. Probst, editors, *Advances in Hardware Design and Verification*, IFIP – Advanced Research Working Conference on Correct Hardware Design and Verification Methods: CHARME'97, Chapman and Hall, 1997.
9. C.J. Lillieroth and S. Singh: Formal Verification of FPGA Cores. *Nordic Journal of Computing* 6, 27-47, 1999.
10. M. Ljung: Formal Modelling and Automatic Verification of Lustre Programs Using NP-Tools, Master's thesis, Prover Technology AB and Department of Teleinformatics, KTH, Stockholm, 1999.
11. M. Sheeran and G. Stålmarck: A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16:1, January 2000.
12. M. Sheeran and G. Stålmarck: Checking safety properties using induction and boolean satisfiability. Appendix to deliverable d20.2, EU project CRISYS, 1999.
13. G. Stålmarck: Stålmarck's Method and QBF Solving. In Proc. Int. Conf. on Computer-Aided Verification, CAV'99, LNCS, Springer-Verlag, 1999.
14. Xilinx: Xilinx IP Center, <http://www.xilinx.com/ipcenter>.

Combining Stream-Based and State-Based Verification Techniques

Nancy A. Day¹, Mark D. Aagaard², and Byron Cook¹

¹ Oregon Graduate Institute, Beaverton, OR, USA
`{nday,byron}@cse.ogi.edu`

² Performance Microprocessor Division, Intel Corporation, Hillsboro, OR, USA
`maagaard@ichips.intel.com`

Abstract Algebraic verification techniques manipulate the structure of a circuit while preserving its behavior. Algorithmic verification techniques verify properties about the behavior of a circuit. These two techniques have complementary strengths: algebraic techniques are largely independent of the size of the state space, and algorithmic techniques are highly automated. It is desirable to exploit both in the same verification. However, algebraic techniques often use stream-based models of circuits, while algorithmic techniques use state-based models. We prove the consistency of stream- and state-based interpretations of circuit models, and show how stream-based verification results can be used hand-in-hand with state-based verification results. Our approach allows us to combine stream-based algebraic rewriting and state-based reasoning, using SMV and SVC, to verify a pipelined microarchitecture with speculative execution.

1 Introduction

Hardware verification techniques can be broadly grouped into those that reason about both the behavior and structure of circuits, and those that reason just about the behavior. Algebraic techniques, such as retiming (e.g., [2, 3]), manipulate the structure of the circuit while preserving its behavior. They have the advantage of being largely independent of the size of the state space. Algebraic techniques often manipulate stream-based models of circuits, i.e., they treat circuits as functions (streams of values). Algorithmic verification techniques, such as model checking [4, 5], verify properties about the behavior of a state-based model, i.e., a state transition system, and have the advantage of being highly automated.

In this work, we bridge the gap between these two forms of models by proving that verification results in the stream-based world correspond to correctness criteria of state-based models. We use O'Donnell's method to provide both stream- and state-based interpretations of circuit descriptions [6]. We use the notation $\{\cdot\}$ for the stream-based interpretation, and $\llbracket \cdot \rrbracket$ for the state-based interpretation. The first contribution of this paper is proving that the behavioral equivalence of the stream-based interpretation of two models, x and y , implies that the

state-based interpretation of x simulates (\leq_s) the state-based interpretation of y :

$$\{x\} = \{y\} \implies \llbracket x \rrbracket \leq_s \llbracket y \rrbracket$$

We refer to this result as the *Verification Correspondence Theorem*. We use Milner’s definition of simulation [24]. In order to prove the Verification Correspondence Theorem we prove a general result about the consistency between the stream- and state-based interpretations. This general result, which we call the *Interpretation Consistency Theorem*, can be used to prove the relationships between other kinds of correctness criteria, such as bisimulation [24, 25].

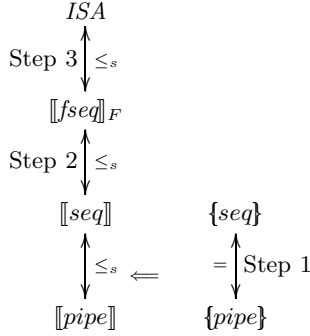


Figure 1. Verification Strategy for the Example

The second contribution of the paper is a demonstration of how the Verification Correspondence Theorem can be used to enable algebraic and algorithmic techniques to work hand-in-hand in microarchitecture verification. An outline of the approach we used for verifying a pipeline against an instruction set architecture (ISA) is illustrated in Figure 1. We decompose the proof that a pipelined microarchitecture with speculative execution ($pipe$) simulates an ISA into three parts. First, we rely on the algebraic manipulation techniques of Matthews and Launchbury to simplify the stream-based interpretation of the pipeline to an equivalent sequential (non-pipelined) model (seq) by retiming and removing forwarding logic ($\{pipe\} = \{seq\}$) [25]. The sequential model is less complex, making it more amenable to automated verification. The Verification Correspondence Theorem allows us to infer that the state-based interpretation of $pipe$ simulates the state-based interpretation of seq ($\llbracket pipe \rrbracket \leq_s \llbracket seq \rrbracket$). Next, we apply the algorithmic techniques of Burch and Dill [24], but first we must overcome the problem that the model we are verifying does not have a flush signal. We construct a version of the model with a flush input ($fseq$), and the second step in the proof uses algorithmic techniques to verify the state-based model without flush ($\llbracket seq \rrbracket$) matches the model with flush when flush is false ($\llbracket fseq \rrbracket_F$) i.e., ($\llbracket fseq \rrbracket_F \leq_s \llbracket seq \rrbracket$). In the third step, we also use algorithmic techniques to check that the state-based interpretation of the sequential model with flush

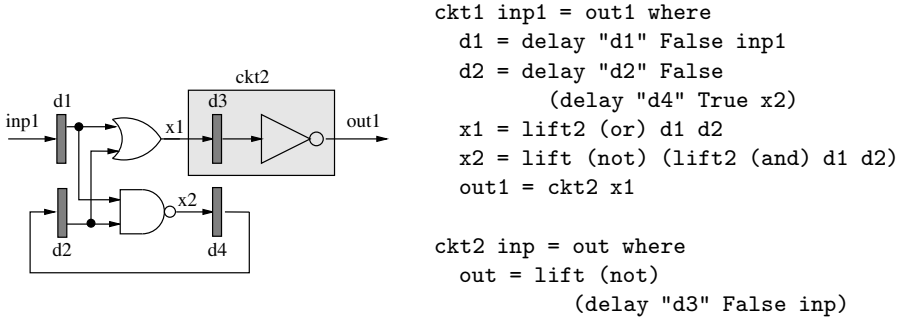
simulates the *ISA* ($\llbracket fseq \rrbracket_F \leq_s ISA$). This proof requires invariants. In the third step, we use symbolic model checking [1] implemented in SMV [2] to check the invariants, and simulation-based validity checking implemented in the Stanford Validity Checker (SVC) [3]. From these three steps, we can conclude that the pipeline simulates the *ISA* ($\llbracket pipe \rrbracket \leq_s ISA$).

Currently, we use separate tools to accomplish each of the tasks in our verification strategy and chain these steps together in a paper proof. Our Verification Correspondence Theorem is designed to facilitate the integration of algebraic and algorithmic reasoning tasks within a theorem proving environment. Towards that end, we have chosen an approach applicable to shallowly embedded models thereby allowing us to use existing algebraic techniques and theorem proving infrastructure. A shallow embedding in a host language or logic means that the primitives of the model are functions of the host language - there is no separate representation of the abstract syntax of the modeling language in the host language [4]. A deep embedding involves an extra level of indirection in proof. Stream- and state-based interpretations of a model are created by providing two different definitions of the base functions for constructing signals. The approach does not require syntactic analysis of the model - it is not a translation - but instead uses the evaluation mechanism of the host language to calculate the two interpretations. It has the advantage that the two interpretations have the same meaning for all constructs except the base functions. We witnessed the benefit of having a shallow embedding in our proof of the Interpretation Consistency Theorem because we could take direct advantage of an existing framework for proving relationships between multiple interpretations [5]. A further benefit of a shallow embedding is that we avoid building the infrastructure of a translator or special-purpose support for the modeling language. Also, the language can use many convenient features of the host language in constructing a model, such as higher-order functions to generate circuits. Our result relating stream- and state- based interpretations could potentially be applied to models written in languages such as Hawk [6, 7], DDD [8], Ruby [9], Lava [10], Lustre [11] and stream-based methods for describing hardware in higher-order logic.

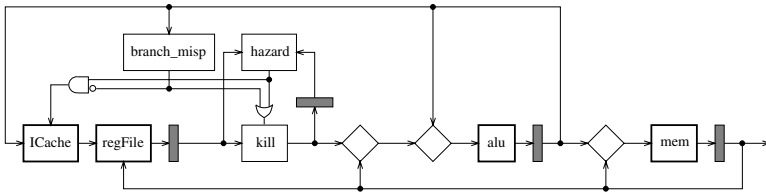
2 Model Descriptions

The base functions for constructing signals are: a family of lift operators (**lift1**, **lift2**, ...) and **delay**. Functions over combinational logic are lifted to functions over signals using **lift**, and latches are introduced using **delay**. The initial value of the latch is supplied as a parameter to the delay operator. Each delay has a string label naming the latch. Circuits are described using a set of possibly mutually recursive equations. Figure 1 shows a simple Boolean circuit. A delay is graphically represented using a narrow shaded box. An advantage of this method of description of hardware is that it facilitates modularity because of the ability to hide internal state.

These base functions form the core of Hawk, a microarchitecture description language shallowly embedded in the pure functional programming language

**Figure 2.** Example circuit

Haskell [14]. Pipelines are constructed using these base functions and Haskell language features such as complex datatypes and higher-order functions. Figure 1 shows a graphical representation of a Hawk pipeline. Hawk uses the *transaction* abstraction to describe a structured collection of data [14, 15]. A transaction contains the opcode, source and destination registers, source and destination values, program counter, and speculative program counter for an instruction. Some units of the pipeline operate on transactions. For example, a register file takes a transaction as input, reads the source values from the register file, and outputs a transaction with these values in the source value fields. A bypass unit (\diamond) checks if the source registers for the incoming transaction match the destination register of the instruction forwarded, and if so, sets the source register values to the value of the destination register. The streams and transactions of Hawk have been used to model superscalar out-of-order microprocessors [14].

**Figure 3.** Hawk pipeline [14]

3 Stream-Based Interpretation and Verification

There are two common, and isomorphic, stream-based interpretations of signals: infinite lists of values and functions from time to values. Our results apply to both representations. Definitions 1-6 are the definitions of the base functions for

the two stream-based interpretations. The stream-based interpretations ignore the label (*label*), which will be used in the state-based interpretation (Section 4).

Streams as infinite lists:

Definition 1 $\{\text{delay}\} \text{ label } i \text{ inp} \triangleq i : \text{inp}$

Definition 2 $\{\text{lift1}\} f \text{ inp} \triangleq \text{map } f \text{ inp}$

Definition 3 $\{\text{lift2}\} f \text{ inp}_1 \text{ inp}_2 \triangleq \text{zipWith } f \text{ inp}_1 \text{ inp}_2$

where $\text{zipWith } z (a : \text{as}) (b : \text{bs}) \triangleq (z \ a \ b) : \text{zipWith } z \ \text{as} \ \text{bs}$
 $\text{zipWith } - \ - \ - \triangleq []$

Streams as functions of time:

Definition 4 $\{\text{delay}\} \text{ label } i \text{ inp} \triangleq \lambda t. \text{ if } t = 0 \text{ then } i \text{ else } \text{inp } (t - 1)$

Definition 5 $\{\text{lift1}\} f \text{ inp} \triangleq \lambda t. f (\text{inp } t)$

Definition 6 $\{\text{lift2}\} f \text{ inp}_1 \text{ inp}_2 \triangleq \lambda t. f (\text{inp}_1 \ t) (\text{inp}_2 \ t)$

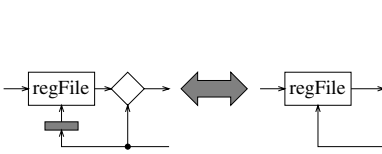




Figure 4. Register Bypass Law 

Matthews and Launchbury derived and verified a set of algebraic rules for simplifying Hawk microarchitecture models using an interpretation of signals as infinite lists . These laws allow the components of a model to be rearranged in behavior-

preserving ways to result in an equivalent but simpler model. For example, their *register-bypass law* (Figure 4) states that a write-before-read register file followed by a bypass, with a delay on its writeback line, has equivalent behavior to a write-before-read register file by itself. Ap-

plication of this rule allows for the removal of forwarding logic once a sufficient amount of retiming has been done in a pipeline. They used Isabelle  both to verify the laws and to transform the pipeline of Figure 5 into the simpler one of Figure 6. While the validity of these rules was proved using coinductive techniques, for the most part the use of the rules only requires rewriting.

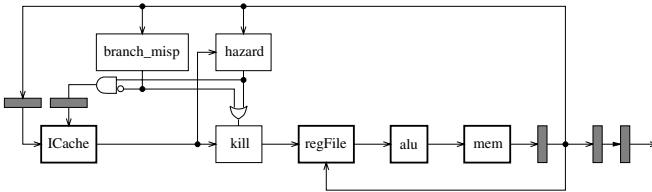



Figure 5. Sequential model resulting from algebraic transformation 

4 State-Based Interpretation and Verification

We use O'Donnell's method of creating a state-based interpretation of a circuit by providing alternative definitions of `delay` and the family of `lift` operators [22]. O'Donnell had multiple interpretations for simulation and generating netlists from a shallowly embedded gate-level hardware description language. The user provides explicit string labels for the delay elements. The addition of the labels provides a means of identifying state-holding elements and detecting feedback loops within a shallow embedding. We extend his method slightly to handle more complex datatypes, such as transactions, in order to apply the technique to microarchitecture verification.

In the state-based interpretation, evaluating a circuit results in a state-transition system: an initial state, a set of next-state equations (one for each latch), and an observation function. A next-state equation matches a string name with combinational circuitry. The observation function is a function that, given a state, computes the values for the output signals. We use N_M , I_M , and O_M to be the next-state equations, initial state and observation function for state model M . The next-state equations and observation functions are represented using a simple datatype for quantifier-free first-order logic. As in the stream interpretation, evaluating a circuit with a combinational loop will not terminate.

Definitions 7 and 8 show the state-based interpretations of `delay` and `lift2`. The initial value (i) for a delay element is provided as one of the arguments. The user supplies a name for the element in the label argument ($label$).

Definition 7 (State-Based Interpretation of Delay).

$$\begin{aligned} \llbracket \text{delay} \rrbracket \text{ label } i \text{ inp} &\triangleq \\ \lambda \text{ seen. if } \text{label} \in \text{seen} & \\ \text{ then } (\emptyset, \emptyset, \text{label}) & \\ \text{ else let } (N, I, O) = \text{inp } (\text{seen} \cup \{\text{label}\}) \text{ in} & \\ \quad (\{(label, O)\} \cup N, \{(label, i)\} \cup I, \text{label}) & \end{aligned}$$

Definition 8 (State-Based Interpretation of lift2).

$$\begin{aligned} \llbracket \text{lift2} \rrbracket f \text{ inp}_1 \text{ inp}_2 &\triangleq \\ \lambda \text{ seen. let } (N_1, I_1, O_1) = \text{inp}_1 \text{ seen} & \\ \quad (N_2, I_2, O_2) = \text{inp}_2 \text{ seen in} & \\ (N_1 \cup N_2, I_1 \cup I_2, f O_1 O_2) & \end{aligned}$$

The initial state, next-state equations and observation function for the simple circuit of Figure 2 produced by the state-based interpretation is:

State variables = d1, d2, d3, d4
 $I_{ckt} = d3 \leftarrow \text{False}, d1 \leftarrow \text{False}, d2 \leftarrow \text{False}, d4 \leftarrow \text{True}$
 $N_{ckt} = d3 \leftarrow d1 \vee d2, d1 \leftarrow \text{inp1}, d2 \leftarrow d4, d4 \leftarrow \neg(d1 \wedge d2)$
 $O_{ckt} = \neg d3$

The state-based interpretation is calculated by passing as arguments a symbolic representation of the inputs along with an empty list as the list of *seen* labels to

the circuit. The calculation follows the data flow backwards through the fanin of each circuit element. When traversing the example circuit, we encounter **d3**, **d1**, and **d4** each once, and **d2** twice. We see **d2** in the fanin of **d3**, and again after having passed through **d2** and **d4**. At each latch, if the label has not been seen, the method retrieves the set of next-state equations, initial state, and combinational circuitry for the inputs. The next-state equation for a latch is the combinational circuitry of the input signal. The combinational output of a delay is just a term representing the name of the delay. If the label has been seen at a latch, as in the case of seeing **d2** a second time when calculating the input for **d4**, the method does not traverse through **d2** again, because it has already been traversed.

The next-state equations resulting from a state-based interpretation of a circuit are used to create input for existing state-based verification tools. We provide links to two such tools: the SMV model checker [10], and SVC, a tautology checker for a subset of first-order logic [11]. Depending on the logic of the tool, we can generate either interpreted or uninterpreted functions for some operators. To link with SMV, we generate an SMV input file. To link with SVC we symbolically simulate the next-state equations in Haskell, and use a previously implemented tight link between Haskell and SVC to create the internal SVC representations of the terms [12].

4.1 Circuit Structure

Because the labels make the structure of the circuit explicit, two circuits with the same behavior in the stream-based interpretation may result in different state-based interpretations. For example, the circuit of Figure 1 has the same observational behavior as that of Figure 2 but the state-based interpretation has the additional state-holding element **d5**. The

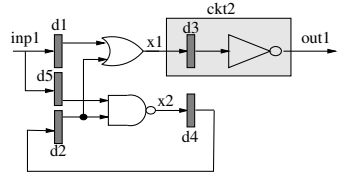


Figure 6. Equivalent to Figure 1

equivalence of the stream-based interpretation of Figures 1 and 2, along with the Interpretation Consistency Theorem ensure both state-based interpretations have equivalent observational behavior.

4.2 Valid Labeling of Delays

A danger in our method is that the user can accidentally use the same label on multiple delays in the circuit. This can lead to an inconsistent interpretation of the model in the state-based world. This *invalid* labeling can occur in two different ways: one that we can detect in the state-based interpretation of **delay** (Definition 2) and one that requires an external design-rule-check for unique labels on all delays.

In Figure 3, the label *c* is used for two different delays. We can detect this kind of invalid labeling because there will be two conflicting next-state equations

for the same label. As a side note, we have an optimized version of `delay` that uses the assumption that the circuit has a valid labeling to avoid redundant traversal.

In Figure 7, both occurrences of `c` appear on the same “branch” of the circuit. When the `c` closest to `a` is encountered, the first `c` is already on the *seen* list, and so the traversal algorithm does not ever encounter `a`. The state-based interpretation of this circuit matches that one shown in Figure 8.

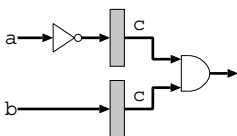


Figure 7. Incorrect labeling 1

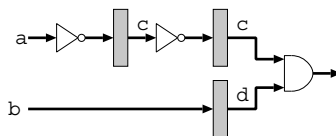


Figure 8. Incorrect labeling 2

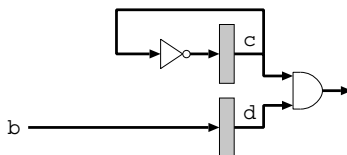


Figure 9. State-based interpretation of Figure 7

4.3 Related Work

Although the aims are the same, our work differs from the signal to state machine translations performed for Lustre by LUSTRE-V2 [10], or for Ruby by T-Ruby [11] and others. Whereas they analyze the program structure and compile a state machine, we want the advantages of working with a shallow embedding and therefore determine the different representations using multiple definitions of base functions.

Singh also uses the technique of alternative definitions for base functions, which he calls *non-standard* interpretations, to provide simulation and test pattern generator interpretations (among others) for Ruby [12]. His work applies only to combinational logic and feedback loops are not allowed.

The original version of the Haskell-based hardware description language Lava was a monadic embedding in Haskell [13]. It used monads to thread the process of generating unique names for state-holding elements through the circuit description in a hidden manner. A difficulty with this approach is that normal function application cannot be used, but rather a new function application operator must be defined.

The recent version of Lava [14] uses Claessen and Sands notion of “observable sharing” [15], which does not require labels for delays, to provide multiple

interpretations of a shallowly embedded language. Their approach is to use a language extension to Haskell that allows them to discover sharing within the heap. The advantage to their approach is that the model does not have to name each state-holding element. The disadvantage is that observable sharing is an impure language feature that precludes the applicability of their approach to pure programming languages and formalisms such as higher-order logic. For generality, we have followed O'Donnell's approach of explicit labeling. In addition to the advantage of generality, we find that explicit labeling has a pragmatic advantage: labels allow the user to provide the names used in the state-based verification. This technique has advantages for debugging and allows the verifier to manage the complexity of many automated verification techniques more easily through means such as controlling variable order. The disadvantage of explicit labeling is that the user has to write these labels and inconsistent labeling is possible. O'Donnell describes methods for generating names semi-automatically, such as using the circuit hierarchy to create structured labels.

5 Formal Connection between State and Stream Interpretations

In this section, we formalize and verify the connection between state- and stream-based verification. Our main result of the section is the Verification Correspondence Theorem (Theorem 1), which says that if two models, x and y , are equivalent in the stream-based interpretation, then the state-based interpretation of x simulates (\leq_s) the state-based interpretation of y .

Theorem 1 (Verification Correspondence).

$$\forall x, y. \{x\} = \{y\} \implies \llbracket x \rrbracket \leq_s \llbracket y \rrbracket$$

For simulation (\leq_s), we use Milner's definition [24], restated as Definition 1, where $x \leq_s y$ means that there exists a relation R , such that if R holds for two states q_x and q_y , then R must hold after taking one step of x and y .

Definition 9 (Simulation).

$$x \leq_s y \triangleq \exists R. \forall q_x, q_y. R(q_x, q_y) \implies R(N_x q_x, N_y q_y)$$

Our proof of the Verification Correspondence Theorem assumes that the two models, x and y , have a valid labeling, as described in Section 4. The proof is described in Section 5 and relies upon our Interpretation Consistency Theorem. Our Interpretation Consistency Theorem shows that O'Donnell's method produces consistent interpretations in the state- and stream-based worlds; its proof is sketched in Section 6.

5.1 Consistency of State and Stream Interpretations

The hardware modeling language we use, Hawk, is implemented as a shallow embedding in Haskell. The fundamental type for describing hardware in Hawk

is a **signal**: wires are **signals** and gates are functions from **signals** to **signal**. Our stream-based interpretation of a signal is a function of time. Our state-based interpretation is a state transition system that consists of an initial state, next-state function, and observation function.

We define *consistency* between state- and stream-based interpretations of a model x to mean that: for all k , iterating the state-based interpretation k times produces the same output as sampling the stream-based interpretation at time k (Theorem 4). The function `StsToStream` iterates a state transition system to produce a stream (Definition 10).

Theorem 2 (Interpretation Consistency).

$\forall x : \text{signal}. \text{StsToStream } \llbracket x \rrbracket = \{x\}$

Definition 10.

$\text{StsToStream } (N, I, O) \ t \triangleq O \ (\text{StsToStream}'(N, I) \ t)$

where $\text{StsToStream}'(N, I) \ 0 \triangleq I$

$\text{StsToStream}'(N, I) \ t \triangleq N \ (\text{StsToStream}'(N, I) \ (t - 1))$

In Hawk, **signal** is a parameterized type (e.g., **signal** α). To simplify the proof of the Interpretation Consistency Theorem, we treat signals as an unparameterized type (**signal**) over an uninterpreted type T . We also restrict ourselves to `lift1`, which lifts functions of one argument (e.g., an inverter). Extending the proof to a parameterized signal type and multi-argument `lifts` would not pose any technical challenges.

Haskell is based on the second-order polymorphic lambda calculus [14]. Our hardware descriptions are programs in an extension of the lambda calculus that includes two new constants: `delay` and `lift`. The proof of our Interpretation Consistency Theorem uses Mitchell and Meyer’s work on *logical relations* [24]. The logical relations framework facilitates proving theorems that relate the meaning of a lambda-calculus program in interpretations that assign different meanings to constants, as we do with `delay` and `lift`.

Mitchell and Meyer’s “Fundamental Theorem of Second-Order Logical Relations” (Theorem 1) states that if a relation, ϕ , is a *constant preserving logical relation* between two interpretations, then the meanings of any program in the two interpretations are related by ϕ . The use of recursion in our hardware descriptions gives us the additional obligation to show that the relation is strict and continuous [24].

Theorem 3 (Fundamental Thm of Second-Order Logical Relations).

For all interpretations $\llbracket \cdot \rrbracket$ and $\{ \cdot \}$ and for all ϕ such that ϕ is a logical relation over $\llbracket \cdot \rrbracket$ and $\{ \cdot \}$: if ϕ preserves the constants of the language then, for all terms e of type τ in the language: $\phi_\tau(\llbracket e \rrbracket, \{e\})$

A logical relation is actually a family of relations: one for each possible type in the language. Mitchell and Meyer introduce and verify a logical relation for the standard types and constants in lambda calculus. Mitchell [24] extends this relation to include recursion.

Our Interpretation Consistency Theorem describes a relationship between the state- and stream-based interpretations of expressions of type **signal**, which is captured in Definition [11](#). We prove the Interpretation Consistency Theorem by 1) extending Mitchell’s logical relation with Definition [11](#) for type **signal** and including **delay** and **lift** as new constants in the lambda calculus; then 2) proving that this new family of relations is a strict and continuous, constant-preserving logical relation.

Definition 11. $\phi_{\text{signal}}(e_1, e_2) \triangleq (\text{StsToStream } e_1 = e_2)$

The three proof obligations of strictness, continuity, and logicity are solved quite easily because we expressed ϕ_{signal} in the lambda calculus. Because ϕ_{signal} fully evaluates both of its arguments, ϕ_{signal} is strict. Only continuous expressions can be expressed in our formalism, and therefore ϕ is continuous. Logicity simply means that ϕ ’s treatment of function application and lambda-abstraction is compatible with the lambda calculus, which it clearly is. To prove that ϕ is constant preserving, we prove that, for all types τ and all constants c of type τ , $\phi_\tau(\llbracket c \rrbracket, \{c\})$.

Theorems [10](#) and [11](#) state that ϕ preserves the meaning of **lift** and **delay**. Of the two, **delay** has the more interesting proof, so we do not describe the verification of **lift**.

Theorem 4 (Consistency of lift).

$$\phi_{(T \rightarrow T) \rightarrow \text{signal} \rightarrow \text{signal}}(\llbracket \text{lift} \rrbracket, \{\text{lift}\})$$

Theorem 5 (Consistency of delay).

$$\phi_{\text{string} \rightarrow T \rightarrow \text{signal} \rightarrow \text{signal}}(\llbracket \text{delay} \rrbracket, \{\text{delay}\})$$

The correctness of **delay** relies on four properties:

1. The value of a label (i.e., a delay element) in a clock cycle is equivalent to evaluating the combinational logic feeding the delay element in the previous clock cycle.
2. If a latch is not used in the circuit (e.g. it has no fanout), then adding the equation for the latch to the next-state equations of the model does not affect the behavior of the model.
3. When evaluating a model, if a delay is encountered whose label is already in the set of next-state equations for the model, then the combinational logic feeding that delay is equivalent to the next-state equation already computed for the label.
4. Every latch in the circuit has an equation in the set of next-state equations of the model.

The first property is proved by induction over the values produced from a delay. The second property relies on the correctness of adding a next-state equation for a label to the set of next-state equations of a circuit. The third and fourth properties are related to the two forms of invalid labelling described in Section [4.2](#). Duplicate labels on different branches of a fanin cone (Figure [10](#)) conflicts with the third property and duplicate labels on the same branch of a fanin cone (Figure [11](#)) conflicts with the fourth property.

5.2 Verification Correspondence

To prove that two models, x and y with equivalent stream-based behavior simulate each other (Verification Correspondence Theorem), we assume $\{x\} = \{y\}$ and prove $\llbracket x \rrbracket \leq_s \llbracket y \rrbracket$.

Applying the Interpretation Consistency Theorem to both sides of the assumption produces:

$$\text{StsToStream } \llbracket x \rrbracket = \text{StsToStream } \llbracket y \rrbracket$$

Using the definition of **StsToStream**, stream equality, and induction, we know that the observations, O_x and O_y , of iterating x and y a total of t times are equal.

$$\forall t. O_x(N_x^t I_x) = O_y(N_y^t I_y)$$

Next we unfold the definition of \leq_s (Definition [4](#)) and provide a witness for the simulation relation. We define two states to be related iff both are reachable in the same number of steps and the observations at those states are equal:

$$R(q_x, q_y) = \exists t. (q_x = N_x^t I_x) \wedge (q_y = N_y^t I_y) \wedge (O_x q_x = O_y q_y)$$

The proof is completed using skolemization, substitution and reasoning about function composition.

6 Example Verification

To demonstrate the use of the Verification Correspondence Theorem, we prove that the state-based interpretation of the pipeline in Figure [1](#) (*pipe*) simulates the *ISA*, which is a state-based model, i.e.:

$$\llbracket \text{pipe} \rrbracket \leq_s \text{ISA}$$

The state-based *ISA* is shown in Figure [2](#). It takes a single input, *stutter*, that we use to make the *ISA* run at the same rate as the pipeline. Figure [3](#) shows the decomposition of our proof. To achieve this result, we chain together proof steps involving two intermediate models, *seq*, and *fseq*. The Verification Correspondence Theorem allows us to use both stream-based algebraic techniques and state-based algorithmic methods in our proof.

The first step in the proof was previously completed by Matthews and Launchbury using algebraic transformations in the theorem prover Isabelle. They showed that the pipeline of Figure [1](#) (*pipe*) is observationally equivalent to the sequential model of Figure [4](#) (*seq*), that is they proved that the stream interpretations of the pipeline and sequential models are equivalent:

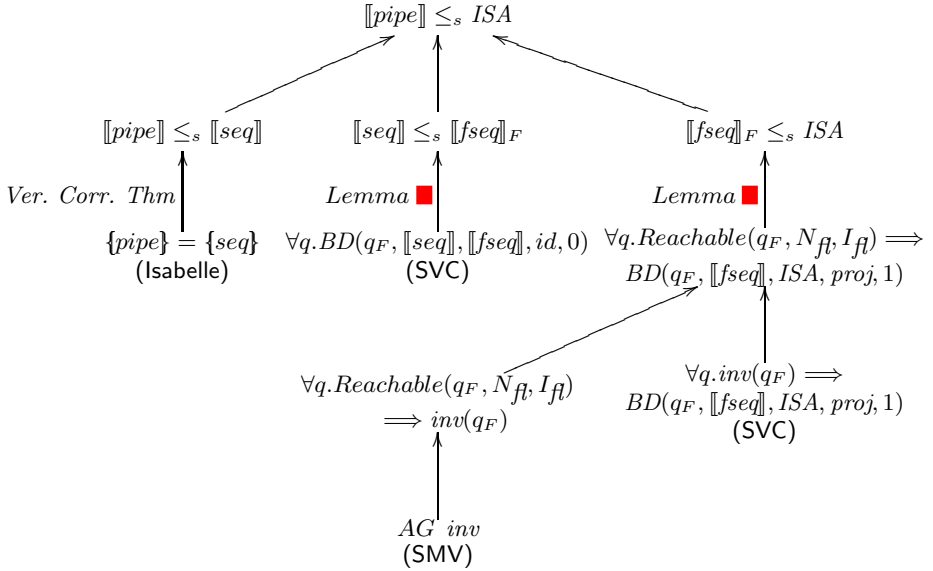
$$\{\text{pipe}\} = \{\text{seq}\}$$

```

 $N_{ISA} =$ 
let (opc, s1, s2, dest, imm) = read instrMem pc
  src1 = read regs s1
  src2 = read regs s2 in
mem  $\leftarrow$  if (( $\neg$ stutter)  $\wedge$  isStore opc)
  then (write mem src1 src2)
  else mem
pc  $\leftarrow$  if stutter
  then pc
  else (if (isIBrz opc) /* indirect branch if zero */
    then (if (src1 = 0) then src2 else (incpc pc))
    else (if (isBrz opc) /* offset branch if zero */
      then (if (src1 = 0) then (incpcby pc imm) else (incpc pc))
      else (incpc pc)))
regs  $\leftarrow$  if ( $\neg$ stutter  $\wedge$  (isAlu opc  $\vee$  isLoad opc))
  then (write regs dest (if (isLoad opc)
    then (read mem src1)
    else (alu opc src1 src2)))
  else regs
instrMem  $\leftarrow$  instrMem

```

Figure 10. ISA



Note: $\llbracket fseq \rrbracket = (N_{fl}, I_{fl}, O_{fl})$

q_F means the state q with the *flush* input False

$\llbracket fseq \rrbracket_F$ means the model *fseq* with the *flush* input False

BD is the Burch-Dill correctness criteria

Figure 11. Proof tree for the example

From this and our Verification Correspondence Theorem, we can conclude the pipeline simulates the sequential model, i.e., $\llbracket pipe \rrbracket \leq_s \llbracket seq \rrbracket$.

We use automated verification techniques applicable to state-based methods to complete the remainder of the proof. For this example, we choose to use SMV and SVC, but other methods could be applied.

Burch and Dill showed that for simple pipelines (such as our sequential one) a simulation relation R can be automatically determined by flushing the pipeline (fl) for some number of steps (n), and projecting relevant state-holding elements (p), i.e., $R(q_x, q_y) \triangleq (\exists n. q_y = (p \circ fl^n)(q_x))$. Substituting this definition of R into the definition of simulation, reduces the task of showing model x simulates model y to showing that for some n :

$$\forall q. N_y((p \circ fl^n) q) = (p \circ fl^n)(N_x q)$$

This is often calling the Burch-Dill commuting diagram. We abbreviate this expression using $BD(q, N_x, N_y, p, n) \triangleq (N_y((p \circ fl^n) q) = (p \circ fl^n)(N_x q))$. The Burch-Dill result is that:

Lemma 1 (Burch-Dill Implies Simulation).

$$\forall x, y. (\forall q. BD(q, N_x, N_y, p, n)) \implies x \leq_s y$$

The flush (fl) operation flushes the pipeline using the next state function N and is defined as:

$$fl\ q \triangleq Nq_T$$

where q_T is the state q with the flush input having value True. To apply the Burch-Dill method, we needed to introduce another intermediate model, called $fseq$, which is a version of the sequential model that takes a flush input. Determining the appropriate behavior for completing a flush and resuming normal operation in the midst of potential hazards and branch mispredictions was one of the more difficult parts of this verification effort. To ensure that $fseq$ has exactly the same behavior as seq when the flush signal is low, we verified that:

$$\forall q. BD(q_F, \llbracket seq \rrbracket, \llbracket fseq \rrbracket, id, 0)$$

where id is the identity function, and q_F is the state q with the flush input having value False. We used SVC to carry out this proof. Based on Lemma 1, from this we can conclude that $\llbracket seq \rrbracket \leq_s \llbracket fseq \rrbracket_F$, where $\llbracket fseq \rrbracket_F$ is the state-based interpretation of the $fseq$ model with the flush input having value False.

The third branch in the proof shows that $\llbracket fseq \rrbracket_F \leq_s ISA$. Again, we use Burch and Dill's result with a projection function ($proj$) that mimics the calculation of the current pc in $fseq$ based on the flush, hazard, and mispredict inputs to map the $fseq$'s pc to the ISA . We found that to prove the commuting diagram, we needed a number of invariants (inv) capturing aspects of the reachable state space. Reachability is defined as $Reachable(q, N, I) \triangleq \exists t. q = N^t I$. We proved,

$$\forall q. Reachable(q_F, N_{fl}, I_{fl}) \implies BD(q_F, \llbracket fseq \rrbracket, \llbracket ISA \rrbracket, proj, 1)$$

in two steps. First, we used SMV to prove the invariants for limited word size and register file size. Second, we use SVC with the ALU and branch prediction units

uninterpreted, to prove that the commuting diagram holds, under the invariants. These two steps are related to simulation using the following lemma, which is easily proven by incorporating reachability into the simulation relation:

Lemma 2 (Burch-Dill with Reachability Implies Simulation).

$$\forall x, y. (\forall q. \text{Reachable}(q, N_x, N_y)) \implies \text{BD}(q, N_x, N_y, p, n) \implies x \leq_s y$$

7 Conclusions and Future Work

The first contribution of this paper is the Verification Correspondence Theorem, which relates stream-based algebraic verification results to Milner’s state-based simulation correctness criteria. This theorem allows reasoning over both interpretations of models to contribute to a verification result. The value of the Verification Correspondence Theorem derives from the complementary strengths of the two approaches: algebraic techniques can handle large state spaces, and algorithmic techniques are largely automated. It has wide application to stream-based modeling languages such as Hawk, DDD, Ruby, Lava, Lustre, and descriptions of hardware in higher-order logic.

The proof of the Verification Correspondence Theorem required a general result about the correspondence between O’Donnell’s stream- and state-based interpretations of models. We plan to investigate how this general result, which we called the Interpretation Consistency Theorem, can be used for non-deterministic models, and stronger correctness criteria such as bisimulation. We also plan to explore how to import state-based results into the stream-based algebraic world, i.e., reversing the implication in the Verification Correspondence Theorem.

Our second contribution is showing the practical application of the Verification Correspondence Theorem in the verification of a pipelined microarchitecture with speculative execution. This example verification pulled together proof steps carried out in the Isabelle theorem prover, the SMV model checker, and SVC. Previously, Matthews and Launchbury’s microarchitectural algebra had no connection to standard state-based techniques for processor verification. We are working on additional connections to techniques such as symbolic trajectory evaluation [30].

Our result is a key ingredient towards creating an integrated theorem-proving verification environment where both stream- and state-based verification techniques work hand-in-hand. We have chosen an approach applicable to shallowly embedded models to simplify the application of existing algebraic techniques and theorem proving infrastructure. Currently, the proof steps in our example are chained together on paper using rules such as transitivity of simulation. Mechanization of these steps within a theorem proving environment would help for proof management and security. The first step in creating an integrated environment is to mechanize the proofs of the Interpretation Consistency and Verification Correspondence Theorems. By working within a theorem proving environment, we could also use our result about the consistency between interpretations to link stream-based algebraic results with state-based algebraic results. For example, recent work by Gordon [15] uses algebraic manipulation of state-based models

to reduce the size of the state space before applying algorithmic techniques such as model checking.

Acknowledgments

We thank Robert Jones, John Launchbury, John Matthews, and John O'Leary, as well as members of PacSoft at OGI, for discussions on this topic. We also thank the FMCAD reviewers and Per Bjesse for their helpful suggestions of improvements. John Matthews supplied three figures describing the Matthews and Launchbury work. Support for this work was provided by Intel, NSF (EIA-98005542), USAF Air Materiel Command (F19628-96-C-0161), and the Natural Science and Engineering Research Council of Canada (NSERC).

References

1. M. D. Aagaard and M. E. Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design (TPCD)*, pages 13-32, Springer, 1994.
2. S. Bainbridge, A. Camilleri, and R. Fleming. Theorem proving as an industrial tool for system level design. In *Theorem Provers in Circuit Design (TPCD)*, pages 253-274. Elsevier Science Publishers, 1992.
3. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD*, volume 1166 of *LNCS*, pages 187-201. Springer, 1996.
4. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming (ICFP)*, pages 174-184. ACM Press, 1998.
5. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design (TPCD)*, pages 129-156, Elsevier, 1992.
6. J. R. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
7. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 68-80. Springer, 1994.
8. K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, 1999.
9. K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. April 9, 2000.
10. E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, April 1986.
11. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, 1998.
12. N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, October 1999.

13. M. Gordon. Reachability programming in Hol98 using BDDs. To appear in *13th International Conference on Theorem Proving and Higher Order Logics (TPHOLs)*, August, 2000.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
15. S. D. Johnson, B. Bose, and C. D. Boyer. A tactical framework for digital design. In *VLSI Specification, Verification and Synthesis*, pages 349–384. Kluwer, 1988.
16. G. Jones and M. Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, 1990.
17. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.
18. J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 288–300. Springer, 1999.
19. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.
20. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.
21. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
22. J. C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.
23. J. C. Mitchell and A. R. Meyer. Second-order logical relations (extended abstract). In *Logic of Programs*, volume 193 of *LNCS*, pages 225–236. Springer, 1985.
24. J. O'Donnell. Generating netlists from executable functional circuit specifications in a pure functional language. In *Functional Programming Glasgow, Workshops in Computing*, pages 178–194. Springer, 1992.
25. D. Park. Concurrency and automata on infinite sequences. In *5th GI Conference on Theoretical Computer Science*, volume 104 of *LNCS*. Springer, 1981.
26. L. C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Lab, 1993. Latest edition: 24 November 1997.
27. J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.4)*. Yale University, Department of Computer Science, RR-1106, February 1997.
28. J. Sawada and W. Hunt. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 364–375. Springer, 1997.
29. J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. In *Designing Correct Circuits*, 1992.
30. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, March 1995.
31. R. Sharp. T-Ruby: A tool for handling Ruby expressions. August, 1996.
32. S. Singh. Implementation of a nonstandard interpretation system. In *Functional Programming, Glasgow, Workshops in Computing*, pages 206–224. Springer, 1989.
33. P. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA89)*, pages 347–359, 1989.

A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles^{*}

Kavita Ravi¹, Roderick Bloem², and Fabio Somenzi²

¹ Cadence Design Systems
kravi@cadence.com

² University of Colorado at Boulder
{Roderick.Bloem, Fabio}@Colorado.EDU

Abstract. Detection of fair cycles is an important task of many model checking algorithms. When the transition system is represented symbolically, the standard approach to fair cycle detection is the one of Emerson and Lei. In the last decade variants of this algorithm and an alternative method based on strongly connected component decomposition have been proposed. We present a taxonomy of these techniques and compare representatives of each major class on a collection of real-life examples. Our results indicate that the Emerson-Lei procedure is the fastest, but other algorithms tend to generate shorter counter-examples.

1 Introduction

Model checking of specifications written in temporal logics such as LTL and fair CTL, or specified as ω -automata involves finding *fair cycles*. Since most model checking tools use one of these specification formalisms [16, 17, 1], detecting fair cycles is at the heart of model checking.

The most popular type of ω -automaton is the (generalized) Büchi automaton [2]. The fairness condition of a Büchi automaton consists of several sets of accepting states. A run is accepted if contains at least one state in every accepting set infinitely often. Consequently, the language of the automaton is nonempty iff the automaton contains a *fair cycle*: a (reachable) cycle that contains at least one state from every accepting set, or, equivalently, a *fair strongly connected component*: a (reachable) nontrivial strongly connected component (SCC) that intersects each accepting set.

LTL model checking [15, 17], language containment based on L -automata [17], and CTL model checking with fairness constraints [1] all reduce to checking emptiness of the language of a Büchi automaton. Hence, the core procedure in many model checking algorithms is to find a fair SCC if one exists, or to report that no such SCC exists.

In typical applications, existence of an accepting run indicates failure of the property. In this case, it is essential that the user be given an accepting run as a *counter-example*, typically presented in the form of a finite *stem* followed by a finite cycle. The counter-example should be as brief as possible, to facilitate debugging. Finding the shortest counter-example, however, is NP-complete [11, 2].

^{*} This work was supported in part by SRC contract 98-DJ-620 and NSF grant CCR-99-71195.

The traditional approach to determine the existence of a fair SCC is to use Tarjan’s algorithm [19]. This algorithm is based on depth-first search and runs in linear time in the size of the graph. In order to do the depth-first search, it manipulates the states of the graph *explicitly*. Unfortunately, as the number of states variables grows, an algorithm that considers every state individually quickly becomes infeasible.

Symbolic algorithms [20] manipulate sets of states via their characteristic functions. They derive their efficiency from the fact that in many cases of interest large sets can be described compactly by their characteristic functions. In contrast to explicit algorithms, an advantage of symbolic algorithms, which typically rely on breadth-first search, is that the difficulty of a search is not tightly related to the size of the state space, but is more closely related to the diameter of the graph and the size of the symbolic representation.

Several symbolic algorithms have been proposed that use breadth-first search and compute a set of states that contains all the fair SCCs, without enumerating them [1, 21, 22, 23]. We refer to such sets as *hulls* of the SCCs. A *SCC-hull* algorithm is one that returns an empty set when there are no fair SCCs and computes a SCC-hull otherwise.

Recently, a symbolic approach for SCC enumeration has been proposed [24]. This algorithm explicitly enumerates SCCs, but not states. The technique is based on the observation that the SCC containing a state v is the set of states with both a path to v and a path from v . The earliest application of this observation to symbolic algorithms known to these authors is in the witness generation procedure of [25]. Xie and Beereel have proposed its use as the primary mechanism to identify SCCs. A similar algorithm, with a tighter bound on complexity, is proposed in [26]. We refer to these algorithms as *symbolic SCC-enumeration algorithms*.

Cycle-detection algorithms have mostly been presented in isolation, with their own proofs of correctness and limited comparison to other algorithms. In Section 1, we shall provide a comprehensive framework for SCC-hull algorithms, which is used to characterize the possibilities for cycle-detection algorithms, and to show correctness for any algorithm in this class. We discuss the relative merits of the currently known cycle-detection algorithms in Section 1 and evaluate their efficiency on a set of practical examples in Section 1. We report two important measures: The time it takes to decide whether a fair cycle exists, and the length of the counter-example that is offered to the user for debugging. We conclude the paper with Section 1.

2 Preliminaries

A graph is a pair $G = (V, E)$, with V a finite set of *states*, and $E \subseteq V \times V$ the set of *edges*. A *path* from $v_1 \in V$ to $v_k \in V$ is a sequence $(v_1, \dots, v_k) \in V^+$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. If a path from u to v exists, we say that u *reaches* v . The *distance* from u to v is the length of a shortest path from u to v if u reaches v ; infinity otherwise. The *diameter* d of G is the largest finite distance between two states in V , and n is the number of states.

A *strongly connected component* (SCC) of G is a maximal set of states $U \subseteq V$ such that for each pair $u, v \in U$, u reaches v . An SCC U is *trivial* if the subgraph of G induced by U has no edges. A set $V' \subseteq V$ is *SCC-closed* if every SCC that intersects V' is wholly contained in it.

The SCCs of G (both trivial and non-trivial) partition V into equivalence classes. Let $[v]$ denote the SCC U such that $v \in U$. The *SCC quotient graph* is a graph (W, H) , such that W is the set of the SCCs of G , and $(U_1, U_2) \in H$ if and only if $U_1 \neq U_2$ and there exist $u \in U_1, v \in U_2$ such that $(u, v) \in E$. We define h to be the length of the longest path in the SCC quotient graph.

The quotient graph is directed and acyclic; hence it defines a partial order on the SCCs such that $[u] \leq [v]$ if and only if u reaches v . The sources of the graph are the minimal or *initial* SCCs, while the sinks of the graph are the maximal or *terminal* SCCs. The length of the longest reachable path through the SCC quotient graph is denoted by h . The number of reachable SCCs in a graph is denoted by N and the number of reachable non-trivial SCCs by N' . Note that $h \leq N$.

Let $C = \{c_1, \dots, c_m\} \subseteq 2^V$ be a set of *Büchi fairness conditions*. An SCC U is *fair* if it is not trivial and $U \cap c_i \neq \emptyset$ for $1 \leq i \leq m$.

Given some encoding of the set of states, *symbolic graph algorithms* operate on the characteristic functions of sets of states and sets of edges. They achieve efficiency by manipulating many states at once. In particular, they can compute all the successors and predecessors of a set of states in a graph in time and space that depends more on the sizes of the characteristic functions than on the number of states. Each such computation is called a *step*. Computing all successors or predecessors in a single operation naturally leads to the use of breadth-first search (BFS) in symbolic algorithms. Depth-first search, by contrast, is not well-suited to symbolic implementation.

Throughout the paper, complexity bounds will be given in steps. Relating the maximum number of steps to the number of states n is not very indicative of performance for practical model checking experiments, for which n can be very large. Hence, we shall concentrate on bounds restricted to the quantities $d, h, |C|, N, N'$, and $|W|$.

2.1 The Propositional μ -Calculus

Symbolic computations are conveniently described by formulae of the μ -calculus [1, 2], which is obtained from first-order predicate logic by adding the least (μ) and greatest (ν) fixpoint operators. Let \mathcal{V} be a set of variables and A a set of atomic propositions. We use the following syntax, which corresponds to the propositional μ -calculus.

- If $f \in A \cup \mathcal{V}$, then f is a formula;
- If $f \in A$, then $\neg f$ is a formula;
- if f and g are formulae, so are $f \wedge g, f \vee g, EX f, EY f, AX f$, and $AY f$;
- if $Z \in \mathcal{V}$ and f is a formula, then $\mu Z.f$ and $\nu Z.f$ are formulae.

The semantics of μ -calculus are defined with respect to a labeled graph (G, L) , where $G = (V, E)$ is a graph and $L : V \rightarrow 2^A$ is a labeling function.

Let $\mathcal{E} = \{e : \mathcal{V} \rightarrow 2^V\}$ be the set of *environments*, that is, the set of functions that associate a set of states to each variable, and let Φ be the set of μ -calculus formulae over A and \mathcal{V} . Then function $\mathcal{I} : \Phi \times \mathcal{E} \rightarrow 2^V$ associates a set of states to a formula $\varphi \in \Phi$ in a given environment $e \in \mathcal{E}$. Let $e[V'/Z]$ be the environment that coincides with e ,

except that $e[V'/Z](Z) = V'$. Function \mathcal{I} extends \mathcal{E} in the natural way, where

$$\begin{aligned}\mathcal{I}(\text{EX } f, e) &= \{v \in V : \exists(v, v') \in E, v' \in \mathcal{I}(f, e)\}, \\ \mathcal{I}(\text{AX } f, e) &= \{v \in V : \forall(v, v') \in E, v' \in \mathcal{I}(f, e)\}, \\ \mathcal{I}(\text{EY } f, e) &= \{v \in V : \exists(v', v) \in E, v' \in \mathcal{I}(f, e)\}, \\ \mathcal{I}(\text{AY } f, e) &= \{v \in V : \forall(v', v) \in E, v' \in \mathcal{I}(f, e)\}, \\ \mathcal{I}(\mu Z. f, e) &= \bigcap \{V' \subseteq V : V' \supseteq \mathcal{I}(f, e[V'/Z])\}, \text{ and} \\ \mathcal{I}(\nu Z. f, e) &= \bigcup \{V' \subseteq V : V' \subseteq \mathcal{I}(f, e[V'/Z])\} .\end{aligned}$$

We use the following abbreviations, which correspond to the familiar future and past tense CTL operators:

$$\begin{array}{ll}\text{EG } f = \nu Z. f \wedge \text{EX } Z & \text{EH } f = \nu Z. f \wedge \text{EY } Z \\ \text{E } f \text{ U } g = \mu Z. g \vee (f \wedge \text{EX } Z) & \text{E } f \text{ S } g = \mu Z. g \vee (f \wedge \text{EY } Z) \\ \text{EF } f = \text{E true U } f & \text{EP } f = \text{E true S } f .\end{array}$$

The EX (or *preimage*) operator maps a set of states to the set of their direct predecessors. This corresponds to one step of backward symbolic BFS. Similarly, EY (*image* or a forward step) maps a set of states to the set of all their direct successors. The operators EX, AX, and all the abbreviations defined in terms of them are called *future-tense* or *backward* operators. EY, AY, and all the abbreviations defined in terms of them are called *past-tense* or *forward* operators.

3 Computing the Fixpoints

The evaluation of μ -calculus formulae requires the computation of fixpoints. Of particular interest to us are the following formulae.

$$\nu Z. \text{EX } \bigwedge_{c \in C} (\text{E } Z \text{ U } (Z \wedge c)) \quad (1)$$

$$\nu Z. \text{EY } \bigwedge_{c \in C} (\text{E } Z \text{ S } (Z \wedge c)) . \quad (2)$$

These two equations describe SCC hulls. Specifically, (1) describes the set of states that can reach a fair cycle, and (2) describes the set of states that can be reached from a fair cycle. Both these sets include the states contained in a fair SCC. Moreover, the terminal SCCs included in (1) and the initial SCCs included in (2) are guaranteed to be fair. We generalize (1) and (2) to a family of μ -calculus formulae that can be used to find the SCCs of a graph. In Section 4, we will show that most known SCC-hull algorithms fall within this class.

The method of successive approximations [15, 24] can be used for the direct evaluation of fixpoints of continuous functions. It is often the case, however, that efficiency can be gained by transforming the given formula into an equivalent one for which, for instance, convergence of the approximations is faster.

3.1 Order of Evaluation in Fixpoints

Formulae (1) and (2) have the form

$$\nu Z. \tau_0 \left(\bigwedge_{1 \leq i \leq n} \tau_i(Z) \right) . \quad (3)$$

Each τ_i is monotonic. (That is, $x \leq y$ implies $\tau_i(x) \leq \tau_i(y)$.) In addition, τ_1, \dots, τ_n are downward:

Definition 1. A function $\tau : X \rightarrow X$ is downward if, $\forall x \in X, \tau(x) \leq x$.

We can eliminate the conjunction from (1) thanks to the following result:

Theorem 1. For $T = \{\tau_0, \dots, \tau_n\}$ a set of downward monotonic functions,

$$\nu Z. \tau_0 \left(\bigwedge_{1 \leq i \leq n} \tau_i(Z) \right) = \nu Z. \tau_0(\tau_1(\dots(\tau_n(Z))\dots)) .$$

Proof. (Sketch.) We prove by induction that

$$\bigwedge_{1 \leq i \leq n} \tau_i(Z) \geq \tau_1(\dots(\tau_n(Z))\dots) .$$

Letting $Z_\infty = \nu Z. \tau_0(\bigwedge_{1 \leq i \leq n} \tau_i(Z))$, we can then prove, again by induction, that, for $Z_k \geq Z_\infty$,

$$\tau_0(\tau_1(\dots(\tau_n(Z_k))\dots)) \geq Z_\infty .$$

The desired result follows easily. \square

From (3) we know that for a set T of downward functions there is a unique fixpoint p such that every fair sequence over T has a finite prefix that converges to p . A sequence is fair if every element of T appears infinitely often in it. In our case, τ_i is downward for $i > 0$, but not for $i = 0$. However, if we change (1) as follows,

$$\nu Z. Z \wedge \text{EX} \bigwedge_{c \in C} (E Z \cup (Z \wedge c)) .$$

and take $\tau_0 = \lambda Z. Z \wedge \text{EX} Z$, then we have all downward functions, and we can apply them in any fair way. The transformation is guaranteed to preserve the fixpoint by Lemma 2.12 of [15]. The transformation can also be applied to (2).

The conjunction with Z that is used to turn τ_0 into a downward function is not always necessary, as shown by the following definition and lemma. (Cf. [15].)

Definition 2. A set of states Z is backward-closed if $\text{EF } Z = Z$; Z is forward-closed if $\text{EP } Z = Z$.

Lemma 1. If Z is backward-closed, then $\text{EX } Z \leq Z$ and $\text{EX } Z$ is backward-closed. If Z is forward-closed, then $\text{EY } Z \leq Z$ and $\text{EY } Z$ is forward-closed.

Therefore, if backward (forward) closure is a precondition for a computation of $EX\ Z$ ($EY\ Z$), then the conjunction of the result with Z is redundant.

It is always possible to make a given Z closed by removing edges from the graph. (This is done in [15] to deal with compassion constraints.) When forward (backward) closure is desired, the edges into (out of) states not in Z are removed. However, this is not necessarily better than making operators syntactically downward by adding a conjunction with Z .

3.2 A Generic Algorithm for Symbolic SCC-Hull Computation

We can combine (1) and (4) to compute a tighter SCC hull that contains only states that have a path both to and from a fair SCC.

$$\nu Z. EX EY \left[\bigwedge_{c \in C} (E\ Z\ U\ (Z \wedge c)) \wedge \bigwedge_{c \in C} (E\ Z\ S\ (Z \wedge c)) \right], \quad (4)$$

Formula (4) has the form of (3). Hence, it is possible to apply the operators in any order, provided downwardness is guaranteed. This leads to the generic algorithm of Fig. 1.

```

GSH( $G, I, T_F, T_B$ ) { /* graph, initial states, forward operators, backward operators */
   $Z := EP\ I, \gamma := \emptyset;$ 
  do {
     $\zeta := Z;$ 
     $\tau := FAIR\_PICK(T_F - \gamma, T_B - \gamma);$ 
     $Z := \tau(Z);$ 
  } until (CONVERGED( $Z, \zeta, \tau, T_F, T_B, \gamma$ ))
  return  $Z;$ 
}

CONVERGED( $Z, \zeta, \tau, T_F, T_B, \gamma$ ) {
  if ( $Z \neq \zeta$ ) {
     $\gamma := \emptyset;$ 
    return false;
  } else {
     $\gamma := \gamma \cup \{\tau\};$ 
    return  $T_F \subseteq \gamma \vee T_B \subseteq \gamma;$ 
  }
}

```

Fig. 1. Generic SCC-hull algorithm

In this algorithm, T_F is a set of downward forward operators that consists of $\lambda Z. Z \wedge EY\ Z$ and $\lambda Z. E\ Z\ S\ (Z \wedge c)$ for each $c \in C$. Likewise, T_B is a set of downward backward operators that consists of $\lambda Z. Z \wedge EX\ Z$, and $\lambda Z. E\ Z\ U\ (Z \wedge c)$ for each $c \in C$. Procedure FAIR_PICK guarantees fairness by ignoring operators in γ .

Procedure CONVERGED returns **true** if all operators of either T_F or T_B have been applied since the last time Z changed. Let P , Q , and R be the evaluations of (1) (the set of all states that can reach a fair SCC), (2) (the set of all states that can be reached from a fair SCC), and (3) (the set of states that can both reach and be reached from a fair SCC), respectively. Let \hat{Z} be the evaluation of Z at convergence. Then,

$$R \leq \hat{Z} \leq P \vee R \leq \hat{Z} \leq Q .$$

This implies that \hat{Z} is a hull of the fair SCCs of the graph. The condition that caused convergence (either $T_F \subseteq \gamma$ or $T_B \subseteq \gamma$) determines whether the initial SCCs or the terminal SCCs in \hat{Z} are guaranteed to be fair.

Theorem 2. *GSH takes $O(|C|dN)$ steps.*

The proof follows along the lines of the proof of Theorem 1 in [11]. The next section shows how several practical algorithms can be obtained by choosing a specific strategy for FAIR_PICK. Simple strategies lead to the improved complexity bound of $O(|C|dh)$ steps and simplifications in the convergence.

4 A Taxonomy of Algorithms for Fair Cycle Detection

We are interested in BDD-based fair cycle detection algorithms for their ability to deal with large state graphs (albeit non-uniformly). Our taxonomy of the known algorithms is based on four factors—the μ -calculus representation of the state set, the complexity in terms of number of steps, the relationship between the computed set of states and the SCC quotient graph, and the length of the counter-examples.

If there are fair SCCs, the SCC-hull algorithms may return extra states besides those in the fair SCCs, depending on the specific μ -calculus formulae employed in the computation. The counter-example procedure is tightly connected to the computed state set. When there are fair SCCs in the state graph, the location of the fair SCCs must be known in order to generate the counter-example. In the SCC-enumeration approach, this is obvious since there is only one SCC in question. In the case of the SCC-hull algorithms, the position of the fair SCCs can be characterized depending on the formula applied. Different counter-example generation algorithms can be applied accordingly; the length of the counter-example generated may vary. When using BDDs, the sets explored in breadth-first search, commonly referred to as *onion-rings*, can be used to generate short counter-examples by deriving the shortest paths between pairs of states.

4.1 SCC-Hull Algorithms

In this section, we present a detailed comparison of algorithms that compute a hull, that is, a set of states that contains all fair SCCs. This computation starts with the set of all states and refines it until a fixpoint is reached. The fair SCCs are not enumerated in these algorithms. All but the Transitive Closure algorithm are instances of the generic SCC-hull algorithm of Fig. 1.

Algorithm of Emerson-Lei: The Emerson-Lei algorithm [17] computes the set of all states with a path to a fair SCC by evaluating the nested fixpoint formula

$$\nu Z. \bigwedge_{c \in C} \text{EX}(\text{E } Z \cup (Z \wedge c)) .$$

Each EU ensures the reachability by all states in the current iterate Z of each fair set c . Each application of EX removes some states that do not reach cycles within the current set. The Emerson-Lei algorithm is an instance of the general algorithm presented in Section 3.1 that uses only the backward operators. The schedule of the EX and EU operators in this computation is fixed and fair. The application of the EX and all the EUs in each outer iteration improves the complexity bound of this algorithm to $O(|C|dh)$ steps [17].

The backward operators (EX and EU) compute the set that can *reach* fair SCCs and result in all terminal SCCs being fair. The hull defines a node cut on the SCC quotient graph, passing through fair SCCs, such that no SCC greater than an SCC on the cut is fair. The hull itself is defined by the set of SCCs less than or equal to the SCCs on the cut which may include unfair SCCs (nodes to the left of $\{4, 7, 8\}$ in Fig. 1).

A counter-example procedure for this algorithm was suggested in [4]. Starting at the initial state, a path is found to the closest fair state. The search continues from one fair state to another until the path passes through all fair sets. Then an attempt is made to complete the cycle from the last fair state to the first fair state. If this attempt fails, i.e., if the first and last fair states belong to different SCCs, the procedure is restarted with the last fair state. A counter-example is guaranteed to be found since all terminal SCCs are fair. This algorithm attempts to find a cycle in a fair SCC close to the initial state. However there is no guarantee regarding the proximity to the initial states since terminal SCCs may be far from the initial states. This may lead to very long stems. The authors of [4] are aware of this drawback and leave the evaluation of the trade-off between finding shorter counter-examples and computational expense as an open issue. **Algorithm of Hojati/Kesten:** Hojati et al. [18] (EL2 algorithm) and Kesten et al. [19] provide a fixed-schedule instance of the generalized algorithm of Fig. 1 that uses only forward operators. In our discussion we will refer to this algorithm as the EL2 algorithm. The fixpoint formula restricted to Büchi fairness is

$$\nu Z. \text{EH} \bigwedge_{c \in C} (\text{E } Z \text{ S } Z \wedge c) .$$

Each ES ensures reachability from a fair set within the current set. The ES operator is the temporal dual of the EU computation in the Emerson-Lei algorithm. The outer EH iteration removes states that are not reachable from a cycle within the set. While the forward dual of the Emerson-Lei algorithm has alternating EY and ES operations, the EH operation in the EL2 algorithm provides a different balance than the Emerson-Lei algorithm between reachability from cycles and reachability from fair sets. The different balance unfortunately disallows the $O(|C|dh)$ step complexity enjoyed by the Emerson-Lei algorithm. However, the EL2 method can be shown to perform better than the Emerson-Lei algorithm on some graphs with many trivial SCCs that satisfy all fairness constraints.

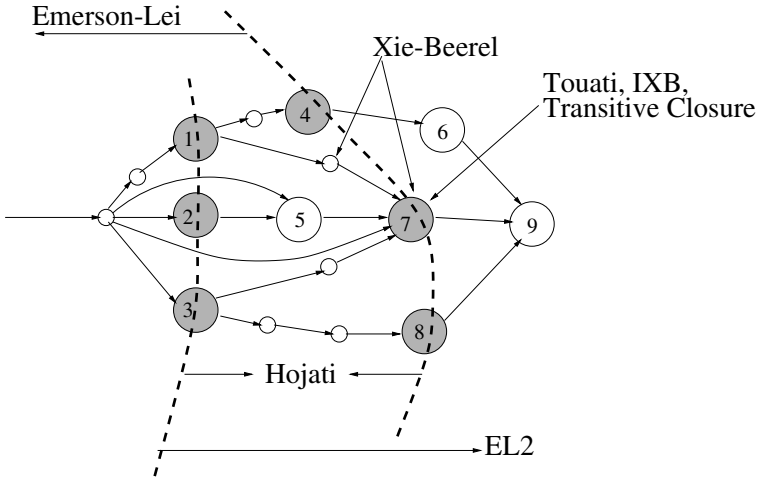


Fig. 2. An SCC quotient graph demonstrating resulting sets of the various methods. Shaded circles are fair SCCs. Small circles are trivial SCCs and large circles are non-trivial unfair SCCs.

The computed hull again defines a cut on the SCC-quotient graph. (Nodes to the right of $\{1, 2, 3\}$ in Fig. 1) The cut passes through a set of fair SCCs such that there are no fair SCCs less than any SCC on the cut. The hull comprises the set of SCCs greater than or equal to those on the cut: The forward operators in the fixpoint algorithm compute a set that can be *reached from* the fair SCCs (the temporal dual of the hull computed by the Emerson-Lei algorithm) and result in all the initial SCCs being fair. The authors of [10] prove that if the operators were EG instead of EH and EU instead of ES, the resulting hull will be the same as in the Emerson-Lei algorithm. This observation is in agreement with Theorem 1.

A counter-example generation procedure for this hull is presented in [11]. The procedure first isolates one initial (fair) SCC in the hull, and then employs the counter-example procedure of [12] to find a path from the initial state to the isolated SCC. The isolation of the fair SCC to determine its location is necessary to apply the counter-example procedure of [12]. A similar approach is used in [13], with the difference that the isolation of the SCC is driven by the distance from the initial states rather than by the position in the quotient graph. It should be noted that an initial fair SCC is not necessarily closer to the initial states than a fair SCC that is not initial. (Node 7 in Fig. 1) In the algorithm of [11] the choice of an initial SCC is mainly dictated by the requirement to deal with compassion constraints (Streett acceptance conditions).

Alternatively, the procedure of [14] can be applied to the hull produced by these algorithms by reversing the direction of edges in the state graph (causing the initial SCCs to become terminal) and picking any state in the set as an initial state. Once the cycle is identified, the stem from this cycle can be traced back to the initial states.

Algorithm of Hojati/Hardin: One of the algorithms presented in [10] and the one in [11] are close variants of each other. We discuss the procedure of [11] and refer to it as the Hojati algorithm. Let

$$EB.f = \nu Z. f \wedge (Z \wedge EY Z) \wedge EX(Z \wedge EY Z) .$$

The fixpoint representation of this algorithm is

$$\nu Z. EB(\bigwedge_{c \in C} Z \wedge EF(c \wedge Z) \wedge EP(c \wedge Z)) .$$

The inner fixpoints ensure reachability of this set *from* and *to* the fair states, while the EB removes states that do not reach or cannot be reached from a cycle within this set. Unlike the previous two methods described here, this computation uses both the forward and backward operators. This procedure is also a fixed-schedule instance of the generalized algorithm of Fig. 1. The schedule is closer to the EL2 algorithm, being more balanced than the Emerson-Lei algorithm in terms of computing reachability to/from cycles and reachability to/from fair sets and consequently cannot have the $O(|C|dh)$ step complexity bound. The algorithm has a stricter convergence criterion than the generalized algorithm. This algorithm is purported by the authors to work best when there are no fair SCCs by converging to the empty set rapidly. The algorithms in [10] and in [11] vary slightly in the schedule in which the operators are applied.

The computed hull induces a subgraph of the SCC quotient graph, which is the intersection of the hulls induced by the Emerson-Lei algorithm and the EL2 algorithm. (Nodes to the right of $\{1, 2, 3\}$ and to the left of $\{4, 7, 8\}$ in Fig. 1.)

COSPAN uses the procedure of [11] to generate counter-examples for the Hojati algorithm. The counter-example procedure presented in [10] may be applied as well.

Transitive Closure Algorithm: Touati et al. developed this symbolic algorithm [12, 13] for the language emptiness check. This algorithm finds the fair SCCs by computing the transitive closure of the transition relation using iterative squaring. (Such a fixpoint computation is not expressible in propositional μ -calculus. It requires $O(\log d)$ transition relation expansions.) This identifies all states that lie on a fair cycle. The resulting set is exactly the set of fair SCCs and is a tighter set than any of the above methods. However, this method is not very efficient when using BDDs and can only be applied to small designs.

Although the fair SCC computation method is inefficient, the counter-example generation procedure proposed for this method is attractive. Touati et al. [12] propose the identification of a fair state (a state contained in some fairness constraint) on an SCC closest to the initial state. To do this they store the set of states reachable from the initial states and the corresponding onion rings. Then they generate the shortest stem to that fair state and complete the cycle through that fair state. The guarantee of the shortest prefix to a fair state on a cycle is possible because the computed fair set is exactly the fair SCCs. For example, this method may first identify 2, 3 or 7 as the fair SCCs in Fig. 1 if one of these SCCs contains a fair state close to the initial state. Consequently, the closest fair state is guaranteed to lie on a fair cycle, although this may not be the fair SCC closest to the initial state. To guarantee a counter-example with a stem of minimal length, it is necessary to find a fair SCC closest to the initial states.

Summary: The SCC-hull algorithms compute sets of states that include all the fair SCCs using fixpoint computations. The Emerson-Lei algorithm has a better complexity bound ($O(|C|dh)$ steps) than the EL2 and Hardin methods ($O(|C|dN)$ steps) although the latter can be shown to perform better than the Emerson-Lei algorithm on some graphs. The hulls computed by the different algorithms and the location of the fair SCCs (important for counter-example generation) are determined by the operators applied in the fixpoint computations. The location of the fair SCCs along with the counter-example generation procedure determines the length of the counter-examples.

4.2 Symbolic SCC-Enumeration Algorithms

In this section, we present methods that enumerate the SCCs of a state graph but do not require explicit manipulation of the states. BDDs can be employed to enumerate the SCCs while taking advantage of the symbolic exploration of the states.

Algorithm of Xie-Beerel: Xie and Beerel [22] propose an algorithm that uses a BDD-based approach in enumerating the SCCs. The main observations that this algorithm uses are that

$$[v] = EF(v) \wedge EP(v) = EF(v) \wedge E(EF(v) \text{ } S \text{ } v) , \quad (5)$$

and that $EF(v) \wedge \neg[v]$ is an SCC-closed set. The algorithm picks a random seed state v from the set of reachable states and computes $EF(v)$ and $E(EF(v) \text{ } S \text{ } v)$. The SCC $[v]$ is computed as defined in 4. If $[v]$ is found to be fair, the algorithm terminates. Otherwise, the remaining states are partitioned into $EF(v) \wedge \neg[v]$ and $V \wedge \neg EF(v)$. The algorithm first recurs on $EF(v) \wedge \neg[v]$ and then on the $V \wedge \neg EF(v)$, restricting the search to the current partition. A pruning procedure FMD_PRED is employed to prune away states in these two partitions that do not reach any non-trivial SCC. The pruning reduces the chances of picking a trivial SCC as the random seed. (The FMD_PRED computation is inefficient since it uses both an EX and an EY computation.) If there are no fair SCCs, all SCCs are eventually enumerated. The complexity of this algorithm is $O(Nd)$ steps. This algorithm can also be applied when the set of reachable states cannot be computed with complexity $O(|W|d)$ steps.

The random selection of a seed state does not provide a systematic enumeration order of the SCCs. For instance, the SCCs marked Xie-Beerel in Fig 4 may be the first ones identified in that graph. A counter-example may be generated by using the method of 12 to generate a prefix to the fair SCC and then finding a fair cycle within it. The length of the counter-example depends on which fair SCC is enumerated first, which cannot be anticipated with this algorithm.

The algorithm in 13 also uses the observation of 4 to compute SCCs. The onion rings for the reachable states are computed incrementally (this method can be applied when the reachable states are too large to compute) and a seed state is picked from the onion-rings starting from the initial states. This is similar to the idea in 12, except that the SCCs are computed as in 5 instead of using the transitive closure of the transition relation. However, the algorithm in 13 does not partition the state space during the search, resulting in $O(Nd)$ steps.

Improvements to the Xie-Beerel Algorithm: We propose two improvements over the Xie-Beerel algorithm that reduce its worst-case complexity and the length of the generated counter-example. The improved Xie-Beerel procedure (IXB) is shown in Fig. 3. The two important enhancements are adding EH to the pruning of the partitions and ex-

```

ENQUEUE( $Q, S, O, M$ ) { /* ENQUEUE (priority queue, set of states, onion-rings, mode) */
  if ( $H \in M$ )  $S := EH\ S$ ;
  if ( $G \in M$ )  $S := EG\ S$ ;
  if ( $S$  does not intersect all fair sets) return ;
   $i :=$  index of the innermost ring,  $O_i$ , that intersects  $S$ ;
  insert ( $S, i$ ) in  $Q$ ; /*  $S$  prioritized by  $i$  */
}

IXB( $R, O$ ) { /* IXB( reachable states, onion rings )*/
   $Q :=$  empty priority queue;
  ENQUEUE( $Q, R, O, \{H, G\}$ );
  while ( $Q$  not empty) {
    ( $V, i$ ) := pop( $Q$ ); /* contains a non-trivial SCC ( $V \neq 0$ ) */
     $v :=$  {random state picked in  $V \cap O_i$ }; /* may be a trivial SCC. */
     $B := E\ V \cup v$ ;
     $S := E\ B\ S\ v$ ;
    if ( $S \neq 0$ )
      if ( $S$  intersects all fair sets) return  $S$ ;
    else  $S := v$ ;
    ENQUEUE( $Q, B \setminus S, O, \{G\}$ ); /* does not need EH */
    ENQUEUE( $Q, V \setminus B, O, \{H\}$ ); /* does not need EG */
  }
}

```

Fig. 3. Pseudocode of the improved version of the Xie-Beerel algorithm. Parameter M to ENQUEUE defines the amount of reduction to be applied to S . Specifying $M = \{H, G\}$ is never wrong, but it may be inefficient. Algorithm IXB uses the onion rings of reachability analysis. It picks the seed in the innermost onion ring that intersects the state space

amining the partitions prioritized by distance from initial state. The first improvement adds EH to the EG operator (corresponding to the FMD_PRED operator in Xie-Beerel). EH prunes states in the partitions that do not reach an SCC in addition to those that are not reachable from an SCC. The worst case complexity of this algorithm can be shown to improve from $O(Nd)$ to $O(\min(N'd + N, N'd + h(N' + 1)))$ due to this addition (Theorem 3 in [14]). The EG operator is more efficient than the FMD_PRED operator that the Xie-Beerel algorithm applies. Also the pruning is controlled by the argument M (mode). Since both modes of pruning (EG and EH) are applied to the initial set (reachable states) and every subsequent partition is either forward or backward closed, pruned

ing only needs to be applied as EH (to forward-closed sets) or EG (to backward-closed sets).

The second improvement over Xie-Beerel is designed to produce shorter counter-examples. A priority queue Q is instantiated to maintain the partitions in the order of their distance from the initial states (computed using the onion-rings). Processing the partitions in the order of their distances from the initial state is guaranteed to yield a fair SCC closest to the initial state. This idea is the same as in [27]. In Fig 4, this method will first remove the initial trivial SCCs and its successor trivial SCC by trimming, and then identify one of 2, 3 and 7 as the fair SCC. The resulting SCC is also the same as the one produced by the algorithm in [27] if the picked seed states are the same, but this algorithm is more efficient since it maintains the partitions. While the Transitive closure procedure does not identify any unfair SCCs, this method may. The number of unfair SCCs examined may be reduced by picking a seed state that is close to the initial states and in many fairness conditions.

Both the original Xie-Beerel procedure and IXB first search backward from the seed state, and then use set B (Fig 5) to trim the forward search. A different algorithm is presented in [28] that uses a lockstep search. Here the forward and backward searches from the seed state are interleaved. The first that terminates is used to trim the other. Since the first that terminates may not be $EF(v)$, IXB and the lockstep search may explore different partitions of the state space. (For more details, refer [28]).

Again, the procedure of [28] can be applied to generate a counter-example. Using the onion rings, the the shortest stem to any fair SCC can be computed. Since the precise location of a fair SCC is known, the complexity of generating the counter-example is reduced [28].

Summary: SCC-enumeration algorithms apply BDD-based approaches to enumerate the SCCs in the graph and apply pruning to reduce the number trivial SCCs examined. Unlike the SCC-hull approaches, fairness of the set being computed is checked after each SCC is identified. Some optimizations have been applied to improve the likelihood of examining fair SCCs, such as discarding partitions that do not intersect one or more fairness constraints. At an additional computational cost, the partition may be refined to contain only states that reach all fairness constraints.

The best SCC-hull algorithm has a $O(|C|dh)$ step bound, whereas the best symbolic SCC-enumeration algorithm has a $O(\min(N'd + N, N'd + h(N' + 1)))$ step complexity bound. The SCC-hull approach may perform better if the diameter is small, whereas the SCC-enumeration approach relies on a small number of (unfair) SCCs to outperform the SCC-hull algorithms. Apart from the worst case complexity, the performance of the algorithms is heavily dependent on the size of the BDDs that arise during the computations, which means that trends are not easily discerned. Experimental results are presented in the following section.

If some model-checking runs are repeated as in a regression test, one may have a pretty good idea of what approach will work better. In particular, if there are no fair cycles, an SCC-hull algorithm may be preferable to SCC enumeration. The expected result may be used to bias the decision between the two approaches.

Table 1. Comparison of computation times and number of steps for Emerson-Lei, EL2, Hojati, IBX1 and Lockstep methods. In Column 3, N stands for no fair SCC, F stands for fair SCC found. Reachability times is in seconds

Circuit	State Vars	N/F	Reach Time (sec)	EL		EL2		Hojati		IXB		Lockstep	
				Time (sec)	EX/EY	Time (sec)	EX/EY	Time (sec)	EX/EY	Time (sec)	EX/EY[SCC]	Time (sec)	EX/EY[SCC]
fischer	4	N	0.06	0.48	170	0.43	275	0.47	247	0.42	205[24]	0.42	208[21]
cycle7	8	N	0.01	0.04	61	0.05	45	0.04	50	1.28	558[83]	2.07	926[125]
Abp2	16	N	0.01	20.32	352	16.52	720	30.91	733	58	2506[693]	59	1689[465]
CI	62	N	1989	4.53	5	4.46	5	8.78	12	4.10	5[0]	4.11	5[0]
F1	70	N	258.6	68.59	4	50.34	4	73.82	12	50.81	4[0]	50.48	4[0]
IR	91	N	1448	2.69	5	2.71	5	56.08	12	3.70	5[0]	3.64	5[0]
RR	94	N	66.5	12.97	50	12.54	50	88.10	58	11.61	50[0]	12.25	50[0]
cmr	97	N	103	0.03	2	0.03	2	0.05	10	0.05	2[0]	0.06	2[0]
S111	191	N	259	0.82	4	0.89	5	4.70	13	1.28	6[0]	1.28	6[0]
S195	483	N	222.5	1.89	3	1.74	3	10.47	10	17.30	3[0]	17.44	3[0]
cycle	7	F	0.1	0.6	183	0.6	232	0.77	305	0.3	16[1]	0.4	16[1]
S142	31	F	10.3	4.5	4	13.1	1285	7.5	12	13.2	1285[1]	13.1	1285[1]
S179	32	F	33.1	0.7	19	0.9	44	1.9	26	0.9	42[1]	0.9	42[1]
S192	44	F	1149	2440	204	117.6	484	4671	462	224.7	172[1]	235	172[1]
BC	70	F	15.5	0.2	20	0.5	14	1	24	0.4	13[1]	0.4	13[1]
S118	80	F	119.5	7	3	340.4	35	4917	26	430.4	34[1]	429.5	34[1]
S124	80	F	117.1	7.1	3	338.1	35	4990	26	425	34[1]	429.8	34[1]
WV	92	F	62.9	288.4	6	4807	16	8958	14	5937	14[1]	5779	14[1]
CO	93	F	773.8	198.5	27	9125	174	96.4	38	8755	174[1]	7179	174[1]
S119	93	F	156.1	137.1	16	1144	157	7720	102	564	45[1]	516	45[1]
S120	93	F	160.8	38	9	643.6	128	4050	74	406.4	43[1]	376.1	43[1]
S106	124	F	185	68.9	35	51	174	399	91	75.5	55[1]	76	55[1]

5 Experimental Results

We implemented five of the algorithms discussed in Section 4—Emerson-Lei, EL2, Hojati, IXB, and Lockstep—in Cospan [3]. The objective was to compare and study the performance of these algorithms in checking language emptiness. All methods are implemented using BDDs. We did not consider the transitive closure algorithm in the experimental evaluation since it is infeasible for large examples. The IXB algorithm is regarded as representative of the Xie-Beerel method. The five implemented methods capture the key ideas in the fair SCC computation algorithms discussed in this paper.

The main metrics in this comparison are the time taken to compute the fair set, the number of steps and the length of the generated counter-example. It is also interesting to compare the performance of SCC-hull algorithms against the symbolic SCC-enumeration algorithms in the presence and absence of fair SCCs. Additionally, the factors that are likely to play an important role in BDD-based computations are the order of variables and the number of dynamic reorderings and the size of the BDDs when

the reorderings are triggered, the frequency of garbage collection of BDD nodes, and the availability of previous computed results in the cache and the unique table of the BDD package.

Tables 1 and 2 present results for 22 examples. These are industrial-strength designs of varying sizes; the size reported for each design is for the part relevant to the property being checked. Property satisfaction is verified using a standard language emptiness check. The number of fairness constraints ($|C|$) in these examples varies from 0 to 14. All experiments were conducted on an SGI machine and 2GB memory with standard options and dynamic reordering. The top half of Table 1 reports examples that do not have a fair SCC and the bottom half reports examples that do. Table 2 reports counter-example related data only for the examples that contain a fair SCC.

Table 3 provides a comparison of the times to compute the fair SCCs by the different algorithms. The first four columns indicate the name of the circuit, the number of state variables, whether a fair SCC was present or not, and the time taken to compute the reachable states. The remaining columns provide the time taken to compute the fair SCCs and the number of steps for each algorithm.

The first 10 examples have no fair SCCs. The SCC-hull algorithms compute an empty resulting set. The Emerson-Lei algorithm and the EL2 algorithm have similar performance and are better than the other methods. Although the Hojati method claims to work better on examples that do not contain a fair SCC, these examples show performance worse (by small factors) than the Emerson-Lei and the EL2 algorithms in more than half the cases. The IXB and the Lockstep methods have similar performance, but worse than the Emerson-Lei and EL2 methods. In cycle7 and Abp2, enumeration of a large number of unfair SCCs contributes to slower run times compared to the Emerson-Lei algorithm. In S195, the time difference is due to the different BDD sizes arising in the different computations, and from the additional pruning step (EH) of the IXB and Lockstep methods. The numbers of steps roughly correlate with the computation times of the various algorithms. The computation times are too small to clearly distinguish the merits of applying forward vs. backward operators, the interleaving of EX, EY, EF, and EP operations or computing the reachability from/to a cycle vs. reachability from/to a fair set.

The bottom half of Table 3 reports data for examples with a fair cycle. Here the Emerson-Lei algorithm performs better than other algorithms on most cases. The EL2 method is an order of magnitude slower on 5 of the 12 examples. The longer runtimes compared to the runtimes of the Emerson-Lei algorithm may be attributed to the larger number of steps, resulting in the increased allocation of BDD nodes, dynamic reorderings, and garbage collections. This may be due to the different state sets arising in the forward and backward computations or the interleaving of reachability to a fair set and reachability to a cycle. In the case of S192, although the number of steps is more than twice the number of steps in the Emerson-Lei method, the BDD sizes are smaller and the number of dynamic reorderings and garbage collections are fewer, therefore requiring a much smaller computation time.

The Hojati method has the worst performance on all examples except S142 and CO. Surprisingly, the computation times are often significantly larger even when the number of steps are fewer than, say, the EL2 algorithm. The justification of these data is sup-

Table 2. Comparison of counter example generation times and length for Emerson-Lei, EL2, Hojati, IBX1 and Lockstep methods

Circuit	State Vars	EL		EL2		Hojati		IXB		Lockstep	
		Time (Prefix, Loop)		Time (Prefix, Loop)		Time (Prefix, Loop)		Time (Prefix, Loop)		Time (Prefix, Loop)	
cycle	7	0.01	(7,7)	0.01	(2,7)	0.01	(2,7)	0.01	(2,7)	0.01	(2,7)
S142	31	1.64	(7,512)	1.62	(7,512)	2.56	(7,512)	1.68	(7,512)	1.60	(7,512)
S179	32	0.10	(10,6)	0.09	(10,6)	0.14	(10,6)	0.10	(10,6)	0.12	(10,6)
S192	44	1.38	(48,32)	0.77	(20,32)	1.52	(28,40)	0.79	(20,32)	0.78	(20,32)
BC	70	0.20	(7,2)	0.19	(6,2)	0.29	(8,2)	0.15	(6,2)	0.18	(6,2)
S118	80	0.60	(7,8)	0.82	(7,10)	1.69	(7,10)	0.73	(7,10)	0.71	(7,10)
S124	80	0.60	(7,8)	0.90	(7,10)	1.71	(7,10)	0.74	(7,10)	0.66	(7,10)
WV	92	0.64	(3,4)	1.44	(3,4)	12.05	(3,2)	6.11	(3,2)	6.07	(3,2)
CO	93	2.93	(13,60)	3.22	(13,32)	3.63	(13,46)	3.32	(13,32)	3.35	(13,30)
S119	93	1.62	(7,20)	1.66	(7,8)	2.80	(8,18)	1.63	(7,18)	1.43	(7,16)
S120	93	1.28	(8,18)	1.34	(7,8)	2.46	(9,16)	1.78	(7,18)	1.58	(7,18)
S106	124	3.11	(51,4)	2.84	(45,6)	4.81	(55,2)	2.95	(45,4)	2.89	(45,4)

ported by increased BDD sizes during the computation, more dynamic reorderings and garbage collections of the unique table. Apart from the state sets encountered during the computation, the inefficiency of the Hojati method in the reuse of previously computed results compared to the Emerson-lei or the EL2 method may explain the increased computation times. The inefficiency is due the interleaving of the EX and EY results in the computed table, causing the eviction of many useful results.

The performance of IXB and the Lockstep methods are tightly connected. In both methods, the first SCC found is fair. The speed for these two methods differs significantly only in a few examples. The number of steps performed are also the same. Hence the Lockstep and the IXB methods are most likely performing the same EX and EY operations in finding the fair SCC, albeit in different order. Compared to the Emerson-Lei algorithm, these methods are generally slower.

In summary, the Emerson-Lei method is the fastest and also has a better complexity bound than the EL2 and Hardin methods. Since different sets are computed by the different algorithms, the performance varies with different examples. The table also indicates that the BDD-related factors play an important role in the computation times. When the run times do not correlate with the number of steps, the BDD statistics provide an explanation for the slowdown or speedup.

In Table 2, we present the computation time and the length of counter-examples for the cases in which a fair SCC was found. The time taken to generate a counter-example is small compared to fair SCC computation time. The counter-examples generated by the EL2, IXB, and Lockstep methods have shorter prefixes than those of the Emerson-Lei algorithm. This suggests that the SCC identified is initial as well as close to the initial state. The reduction in a long prefix such as that of S192 is definitely desirable although a cycle of minimal length cannot be guaranteed by any of the algorithms.

In summary, when the counter-example generated by the fastest algorithm, Emerson-Lei, has a long prefix, then one of EL2, IXB, or Lockstep methods can be applied to find a shorter prefix to the fair cycle. Some hit in computation time is expected for this benefit.

6 Conclusions

We have discussed various symbolic algorithms to check for existence of fair cycles. These algorithms fall in two major classes: the SCC-hull and the SCC-enumeration methods. We have presented a framework for the SCC-hull algorithms, that allows us to easily derive other symbolic algorithms for fair-cycle detection, such as a forward version of Emerson-Lei's algorithm or a particular schedule of the generalized algorithm. Moreover, it allows us to quickly infer correctness of new algorithms.

We have presented a taxonomy and comparison of symbolic algorithms, and we have discussed the merits of the different counter-example generation routines.

Finally, we have compared five of the algorithms using examples from practice. We have found that none of the algorithms that have been proposed as alternatives to the one of Emerson-Lei performs better on a significant number of cases. If, on the other hand, the length of the counter-examples for debugging is the primary concern, the EL2, IXB, and Lockstep algorithms often produce better results.

In future work, we want to separate the influence on efficiency due to four factors—forward versus backward or mixed computations, the computation schedule, the counter-example generation routine, the use of don't-cares and the influence of BDD-related issues.

We plan to study a hybrid routine for fair-cycle detection. The hybrid method may split the state space, as the SCC-enumeration routines do, when the BDDs are large. Splitting allows it to simplify both the state set and the BDDs representing the graph. When the BDDs are simple, the procedure may switch to Emerson-Lei.

Acknowledgments

We would like to thank Bob Kurshan for pointing us to this problem and for very useful suggestions, Ron Hardin for his help with Cospan and Peter Beerel for providing us with an early version of his paper.

References

- [1] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In these proceedings.
- [2] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.
- [3] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

- [4] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.
- [5] J. W. de Bakker and D. Scott. A theory of programs. Unpublished notes, IBM Seminar, Vienna, 1969.
- [6] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.
- [7] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.
- [8] R. H. Hardin, Z. Har’El, and R. P. Kurshan. COSPAN. In *Eighth Conference on Computer Aided Verification (CAV ’96)*, pages 423–427. Springer-Verlag, 1996. LNCS 1102.
- [9] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV’97)*, pages 268–278. Springer-Verlag, Berlin, 1997. LNCS 1254.
- [10] R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV ’93)*, pages 41–58. Springer-Verlag, Berlin, 1993. LNCS 697.
- [11] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω -regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.
- [12] L. Kantorovitch. The method of successive approximations for functional equations. *Acta Mathematica*, 71:63–97, 1939.
- [13] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98)*, pages 1–16, Berlin, 1998. Springer. LNCS 1443.
- [14] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.
- [15] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.
- [16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
- [17] D. M. R. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5, 1970.
- [18] F. Somenzi. Symbolic state exploration. *Electronic Notes in Theoretical Computer Science*, 23, 1999. <http://www.elsevier.nl/locate/entcs/volume23.html>
- [19] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.
- [20] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [21] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω -automata using BDD’s. In *1991 International Workshop on Formal Methods in VLSI Design*, Miami, FL, January 1991.
- [22] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω -automata using BDD’s. *Information and Computation*, 118(1):101–109, April 1995.
- [23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.
- [24] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of the International Conference on Computer-Aided Design*, pages 37–40, San Jose, CA, November 1999.

Correctness of Pipelined Machines ^{*}

Panagiotis Manolios

Department of Computer Sciences, University of Texas at Austin
pete@cs.utexas.edu

<http://www.cs.utexas.edu/users/pete/>

Abstract. The correctness of pipelined machines is a subject that has been studied extensively. Most of the recent work has used variants of the Burch and Dill notion of correctness [1]. As new features are modeled, *e.g.*, interrupts, new notions of correctness are developed. Given the plethora of correctness conditions, the question arises: what is a reasonable notion of correctness? We discuss the issue at length and show, by mechanical proof, that variants of the Burch and Dill notion of correctness are flawed. We propose a notion of correctness based on WEBs (Well-founded Equivalence Bisimulations) [2, 3]. Briefly, our notion of correctness implies that the ISA (Instruction Set Architecture) and MA (Micro-Architecture) machines have the same observable infinite paths, up to stuttering. This implies that the two machines satisfy the same $CTL^* \setminus X$ properties and the same safety and liveness properties (up to stuttering).

To test the utility of the idea, we use ACL2 to verify several variants of the simple pipelined machine described by Sawada in [4, 5]. Our variants extend the basic machine by adding exceptions (to deal with overflows), interrupts, and fleshed-out 128-bit ALUs (one of which is described in a netlist language). In all cases, we prove the same final theorem. We develop a methodology with mechanical support that we used to verify Sawada's machine. The resulting proof is substantially shorter than the original and does not require any intermediate abstractions; in fact, given the definitions and some general-purpose books (collections of theorems), the proof is automatic. A practical and noteworthy feature of WEBs is their compositionality. This allows us to prove the correctness of the more elaborate machines in manageable stages.

1 Introduction

The complexity of computing systems has made mechanical verification the preferred way to reason about such systems. In reasoning about computing systems, we start by modeling the system and stating properties of interest. This is inherently an informal process; therefore, care must be taken to get it right. Computing systems vary as do approaches to modeling them, but notions of correctness often do not. For example, there are many different sorting algorithms, but there is one notion of correctness, *viz.*, that the output is an ordered permutation of

^{*} Support for this work was provided by the SRC under contract 99-TJ-685.

the input. This correctness criterion can be thought of as a constraint satisfied only by sorting algorithms. If we give you an algorithm and a proof that it satisfies this constraint, then you do not have to look at the internals of the algorithm to use it with confidence to sort. On the other hand, if the notion of correctness is that the output is ordered, you have to look at the internals to determine if the algorithm sorts and cannot use it with confidence otherwise, *e.g.*, a trivial algorithm that always returns the empty sequence satisfies this constraint. The right notion of correctness allows you to ignore the details of a system, but the wrong notion is useless; therefore, getting the notion of correctness right is of the utmost importance. Similarly, in the case of pipelined machine verification, a notion of correctness can be thought of as a constraint that, given an ISA (Instruction Set Architecture) specification, is satisfied only by “correct” MA (Micro-Architecture) machines. If the constraint allows trivial implementations, it is as useless for specifying the correctness of pipelined machines as the notion of ordered output is for specifying the correctness of sorting algorithms.

Previous approaches to pipelined machine verification have not been stated in these terms. For example, one finds that there are restrictions placed on the types of pipelined machines that can be checked (*e.g.*, machines with a flush instruction), or that the notion of correctness relates inconsistent MA states (*e.g.*, states not reachable from a flushed state) to ISA states, or that there are various conditions, often separated into safety and liveness conditions, that need to be checked, or that as new features are added, new notions of correctness are used. We explored the situation in detail for the BD (Burch and Dill [1]) variant of correctness used by Hunt and Sawada in [2, 3, 4, 5] because of the availability of proof scripts and because of the ubiquity of the BD approach to pipelined machine verification. We found that trivial machines satisfy this notion of correctness; a mechanical proof establishing this is described near the beginning of Section 6. We must point out that the actual machines verified in the above referenced work are not trivial machines. The machines are remarkably complicated and we consider their verification an impressive body of work.

We propose a notion of correctness based on WEBs (Well-founded Equivalence Bisimulations) that amounts to: when viewed appropriately, the pipelined machine and the ISA machine have the same infinite behaviors, up to stuttering. We account for stuttering because we are comparing systems at different levels of abstraction and an atomic step of one system may not be atomic in the other system. We say “viewed appropriately” because the two systems may represent data in different ways, *e.g.*, we show the equivalence of a machine that operates on integers with a machine that operates on bit-vectors. Another reason for the qualifier is that an MA state has more components than an ISA state, *e.g.*, the pipeline. Even when components overlap, they may exhibit different behaviors, *e.g.*, if the pipeline is non-empty, the program counter of an MA state will point several instructions ahead of the last completed instruction. Therefore, we use a *refinement map* or an *abstraction function* that relates MA states to corresponding ISA states. There are many ways in which to choose a refinement map. For example, BD approaches require that the MA machine have a flush instruction

and relate an MA state with the ISA state obtained by flushing the MA state and retaining only the programmer visible components. A description of the BD notion of correctness and a more thorough discussion of the issues is given in Section 1, but, briefly, the main problem is that the MA state obtained from flushing can look very different from the original MA state. To make this point very clear, let PRIME be the system whose single behavior is the sequence of primes and let NAT be the system whose single behavior is the sequence of natural numbers. We do not consider NAT and PRIME equivalent, but using the refinement map from NAT to PRIME that maps i to the i^{th} prime, we can indeed prove the peculiar theorem that NAT and PRIME have the same behaviors. The moral is that we must be careful to not bypass the verification problem with the use of such refinement maps. Refinement maps that leave the important part of the state untouched are best. Refinement maps based on flushing modify important programmer visible components such as the register file; therefore, we find their use objectionable. As further evidence that something strange is happening, note that flushing-based refinements map inconsistent (unreachable) MA states to reachable ISA states. (See Section 2 for details.) To overcome these obstacles we use refinement maps that—to the extent possible—do not alter components of MA states: we map an MA state to the ISA state obtained by retaining the programmer visible components of the committed part of the state. This can be thought of as invalidating the partially executed instructions, which leaves the register file intact, but may change the program counter.

In Section 3 we define stuttering bisimulation, WEBS, and present the relevant theorems. In Section 4 we describe several variants of a simple pipelined machine described by Sawada in [2, 3]. This machine was designed to explain the issues in pipelined machine verification; it is a toy version of the final machine verified in Sawada’s thesis. Since this is the first paper to use WEBS to verify pipelined machines, it seemed sensible to compare our results to existing work. We describe eleven proofs of correctness involving twelve variants of this machine. The variants include exceptions, fleshed-out ALUs, interrupts, and combinations of these features. In all cases, we proved the same final theorem.

Deciding what theorem to prove is only part of the story. The other part is to get the theorem mechanically checked with limited human interaction. In Section 5, we present a methodology for automating proofs of pipelined machine correctness. We have implemented our methodology in ACL2 (see [4, 5, 6, 7]) and have used it to verify the various previously mentioned machines. Given the descriptions of the machines, the refinement maps, and a few definitions, we automatically generate further definitions and proof obligations. If all the proof obligations are discharged by ACL2, correctness has been established. If not, at least a proof outline has been provided. All of the machines we verified were handled using this methodology and no intermediate abstractions were required. Some proof was required, for example, we show that a netlist description of a circuit is actually a 128-bit adder, but we did not have to prove any invariants about the pipeline. We present conclusions and discuss related work in Section 6.

2 Well-Founded Equivalence Bisimulation

2.1 Preliminaries

\mathbb{N} denotes the natural numbers. $\langle Qx : r : b \rangle$ denotes a quantified expression, where Q is the quantifier, x is the bound variable, r is the range of x (true if omitted), and b is the body. Function application is sometimes denoted by an infix dot “.” and is left associative. Function composition is denoted by \circ . The disjoint union operator is denoted by \uplus . For a relation R , we abbreviate $\langle s, w \rangle \in R$ by sRw . A *well-founded structure* is a pair $\langle W, \prec \rangle$ where W is a set and \prec is a binary relation on W such that there are no infinitely decreasing sequences on W with respect to \prec . We abbreviate $((s \prec w) \vee (s = w))$ by $s \preceq w$. From highest to lowest binding power, we have: parentheses, binary relations (e.g., sBw), equality ($=$) and set membership (\in), conjunction (\wedge) and disjunction (\vee), implication (\Rightarrow), and finally, binary equivalence (\equiv). Spacing is used to reinforce binding: more space indicates lower binding.

Definition 1 (Transition System)

A Transition System (TS) is a structure $\langle S, \dashrightarrow, L \rangle$, where S is a non-empty set of states, $\dashrightarrow \subseteq S \times S$ is a left-total (every state has a successor) *transition relation*, and L is a *labeling function* which maps each state to a label.

For TS M , state s (of M), and temporal logic formula f , $M, s \models f$ denotes that f holds at state s of model M . A *path* σ is a sequence of states such that for adjacent states s, u , $s \dashrightarrow u$. A path σ is a *fullpath* if it is infinite. $fp.\sigma.s$ denotes that σ is a fullpath starting at s .

2.2 Stuttering Bisimulation

We define a variant of stuttering bisimulation [1]. This notion is similar to the notion of a bisimulation [2, 3], but allows for finite stuttering, and therefore differs from weak bisimulation [4]. The idea is to partition the state space of a transition system into equivalence classes such that states in the same class have the same infinite paths up to stuttering. We show that the equivalence class obtained by relating ISA states to MA states, where the label of an MA state is determined by a refinement map, is a stuttering bisimulation.

Definition 2 (Match)

Let i range over \mathbb{N} . Let INC be the set of strictly increasing sequences of natural numbers starting at 0; rigorously, $INC = \{\pi : \pi : \mathbb{N} \rightarrow \mathbb{N} \wedge \pi.0 = 0 \wedge \langle \forall i : i \in \mathbb{N} : \pi.i < \pi(i+1) \rangle\}$. The i th segment of fullpath σ with respect to $\pi \in INC$, ${}^\pi\sigma^i$ is given by the sequence $\sigma(\pi.i), \dots, \sigma(\pi(i+1) - 1)$. For $B \subseteq S \times S$, $\pi, \xi \in INC$, $i, j \in \mathbb{N}$, and fullpaths σ and δ , we abbreviate $\langle \forall s, t : s \in {}^\pi\sigma^i \wedge t \in {}^\xi\delta^j : sBt \rangle$ by ${}^\pi\sigma^i B {}^\xi\delta^j$. For $B \subseteq S \times S$, $\pi, \xi \in INC$:

$$match(B, \sigma, \delta) \equiv \langle \exists \pi, \xi : \pi, \xi \in INC : \langle \forall i : i \in \mathbb{N} : {}^\pi\sigma^i B {}^\xi\delta^i \rangle \rangle$$

Definition 3 (*Equivalence Stuttering Bisimulation (ESTB)*)

B is an equivalence stuttering bisimulation on TS $M = \langle S, \dashrightarrow, L \rangle$ iff:

1. B is an equivalence relation on S ; and
2. $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$; and
3. $\langle \forall s, w \in S : sBw : \langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta) \rangle \rangle \rangle$

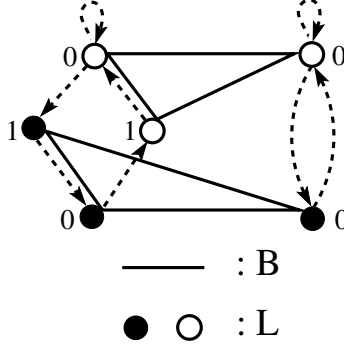


Fig. 1. The graph denotes a transition system, where circles denote states, the color of the circles denotes their label, and the transition relation is denoted by a dashed line. States related by the equivalence relation B are joined by a solid line. Notice that B is ESTB as related states have the same infinite paths, up to stuttering. To check that B is a WEB, let $rank(u, v) = \text{tag of } v$, and use the well-founded witness $\langle rank, \langle \mathbb{N}, < \rangle \rangle$.

An example of an ESTB appears in Fig. 1. ESTBs are the appropriate notion of correctness because if we show that MA and ISA states are related by an ESTB, then it follows that the states have the same behaviors. However, ESTBs are very difficult to prove mechanically because one has to reason about infinite sequences. We would much rather reason about single steps. This is the motivation for the following definition.

Definition 4 (*Well-Founded Equivalence Bisimulation (WEB [14, 15])*)

B is a well-founded equivalence bisimulation on TS $M = \langle S, \dashrightarrow, L \rangle$ iff:

1. B is an equivalence relation on S ; and
2. $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$; and
3. There exists function $rank : S \times S \rightarrow W$, with $\langle W, < \rangle$ well-founded, and

$$\begin{aligned}
 &\langle \forall s, u, w \in S : sBw \wedge s \dashrightarrow u : \\
 &\quad \langle \exists v : w \dashrightarrow v : uBv \rangle \vee \\
 &\quad (uBw \wedge rank(u, u) < rank(s, s)) \vee \\
 &\quad \langle \exists v : w \dashrightarrow v : sBv \wedge rank(u, v) < rank(u, w) \rangle \rangle
 \end{aligned}$$

We call a pair $\langle \text{rank}, \langle W, \prec \rangle \rangle$ satisfying condition 3 in the above definition, a *well-founded witness*. An example of a WEB appears in Fig. 1. Note that to prove a relation is a WEB, reasoning about single steps of \rightarrow suffices.

Notice that the difference between WEBs and ESTBs is in their third conditions. The third WEB condition can be thought of saying that given states s and w in the same class, such that s can step to u , u is either matched by a step from w , or u and w are in the same class and the rank function decreases (to guarantee that w is forced to take a step), or some successor v of w is in the same class as s and the rank function decreases (to guarantee that u is eventually matched). We have the following theorems.

Theorem 1 (cf. [15, 16]) *B is an ESTB on TS M iff B is a WEB on M .*

Theorem 1 says that the notion of stuttering bisimulation is exactly captured by the notion of WEB. The significance is that any stuttering bisimulation can be proved using WEBs.

Theorem 2 (cf. [2, 15]) *If B is a WEB on TS M and sBw , then for any $CTL^* \setminus X$ formula f , $M, s \models f$ iff $M, w \models f$.*

Theorem 2 says that states related by a WEB satisfy the same next-time free formulae of the branching-time logic CTL^* . As a consequence, they satisfy the same LTL (Linear Temporal Logic) formulae and the same safety and progress properties (up to stuttering). This is a strong notion of equivalence and that we can use it profitably depends, in part, on the use of refinement maps, which we now discuss.

2.3 Refinement and Composition



In this section, we define a notion of refinement and show that WEBs can be used in a compositional fashion. The compositionality of WEBs allows one to prove the correctness of a pipelined machine in stages, where at each stage, one can focus on a specific aspect of the machine.

Definition 5 (Refinement)

Let $M = \langle S, \rightarrow, L \rangle$, $M' = \langle S', \rightarrow', L' \rangle$, $r : S \rightarrow S'$. We say that M is a *refinement* of M' with respect to refinement map r if there exists an equivalence relation, B , on $S \uplus S'$ (the disjoint union of S and S') such that $sB(r.s)$ for all $s \in S$ and B is a WEB on the TS $\langle S \uplus S', \rightarrow \uplus \rightarrow', L \rangle$, where $\mathcal{L}.s = L'(r.s)$ for s an S state and $\mathcal{L}.s = L'(s)$ otherwise.

Consider **MA_{net}**, a pipelined machine which represents numbers as bit-vectors and with an ALU described in a netlist language, and **MA**, a pipelined machine which represents numbers directly and whose ALU is described in terms of the familiar arithmetic operators. The labeling function of these two systems is the identity function. To show that **MA_{net}** refines **MA** we exhibit a refinement map which maps **MA_{net}** states to **MA** states by turning bit-vectors into numbers and


prove that the resulting equivalence relation is a WEB. A consequence is that related **MA**_{net} and **MA** states have the same behaviors modulo the representation of numbers.


Theorem 3 (*Composition* (, cf. ))

If $\langle S, \dashrightarrow, L \rangle$ is a refinement of $\langle S', \dashrightarrow', L' \rangle$ with respect to r , and $\langle S', \dashrightarrow', L' \rangle$ is a refinement of $\langle S'', \dashrightarrow'', L'' \rangle$ with respect to q , then $\langle S, \dashrightarrow, L \rangle$ is a refinement of $\langle S'', \dashrightarrow'', L'' \rangle$ with respect to $q \circ r$.

Consider the following application of this theorem. Suppose we prove that **MA**_{net} is a refinement of **MA**, as discussed above. Suppose we also show that **MA** is a refinement of **ISA**, the non-pipelined specification. Using the composition of WEBs theorem, we have that **MA**_{net} is a refinement of **ISA**. An advantage to proving correctness in stages is that if the difference between the machine descriptions from one stage to the next is not too great, we may be able to have the proofs go through automatically. This is because there is enough structural similarity in the two machines that ACL2 can decide equivalence on its own. In our experiments with various versions of Sawada's small machine, this is what we observed. Yet another advantage is that changes to the lower-level machines can be localized. For instance, if we change the adder from a serial adder to a carry-lookahead adder, then the resulting proof obligations are isolated to showing that the carry-lookahead adder is equivalent to the serial adder.

3 Pipelined Machine Verification

In this section, we describe various versions of Sawada's simple pipelined machine  and the correctness criteria proved. We discuss correctness, the deterministic variants, and finally the non-deterministic ones.



Machines are modeled as functions in ACL2, *e.g.*, the first machine we define is **ISA** and this amounts to defining **ISA-step**, a function that given an **ISA** state returns the next state. For all the machines, an instruction is a four-tuple consisting of an opcode, a target register, and two source registers. We give informal descriptions of the machines in this paper, since the formal descriptions in ACL2 are available from the author's Web page .

3.1 Correctness

In the introduction, we made the case that any notion of correctness can be thought of as a constraint. Pipelined machines that satisfy the constraint are "correct" implementations of the **ISA**, with respect to this notion of correctness. We can judge the merits of a notion of correctness by checking that no obviously incorrect machine satisfies the related constraint. We argued that the refinement maps should be understandable, *e.g.*, if we map **MA** states to **ISA** states then there should be a clear relationship between related states. Applying these criteria to the BD variant under consideration, we find that in both respects, this notion of correctness is flawed.

The notion of correctness that we use is simple to state: we show that the pipelined machine is a *refinement* of the ISA specification. Notice that any notion of correctness has to account for stuttering, *e.g.*, a pipelined machine requires several cycles to fill the pipeline, whereas an ISA machine executes an instruction per cycle; any notion of correctness also has to account for refinement, *e.g.*, a pipelined machine may represent numbers as bit-vectors, whereas the ISA machine may represent them directly. Our notion of correctness is the strongest notion that we can think of which accounts for stuttering and refinement. We will discuss the verification of a number of machines, but, regardless of the proof details, we always prove the same theorem, *viz.*, that a pipelined machine has the same infinite paths (up to stuttering and refinement) as an ISA machine.

3.2 Deterministic Machines

The deterministic machines are named **ISA**, **MA**, **ISA128**, **MA128**, **MA128serial**, and **MA128net**. **ISA** and **MA** correspond to the machines in  and the simple machines in . We start with descriptions of **ISA** and **MA** and compare the BD approach to correctness with ours.

ISA. An **ISA** state is a three-tuple consisting of a program counter (pc), a register file, and a memory. The next state of an **ISA** state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, subtraction, and noop. In the case of addition, the target register is updated with the sum of the values in the source registers and the program counter is incremented. Subtraction is treated similarly. In the case of a noop, only the program counter is incremented. **ISA** is the specification for **MA**.

MA. **MA** is a three-stage pipelined machine. An **MA** state is a five-tuple consisting of a pc, a register file, a memory, and two latches. The three stages are the fetch stage, the set-up stage, and the write-back stage. During the fetch stage, instructions are fetched from memory and stored in the first latch; during the set-up stage, the instruction in the first latch is passed to the second latch, but with the values of the source registers; during the write-back stage the values and the opcode are fed to the ALU which performs the appropriate instruction and the result is written into the target register, if the instruction was not a noop. The **MA** machine can execute one instruction per cycle once the pipeline is full, except when there are successive arithmetic instructions where the second instruction uses the target register of the first instruction as a source register. In this case, the machine is stalled for one cycle in order for the target register to be updated.

One difference between **MA** as defined above and the version given by Sawada (**SMA**) is that **SMA** has an extra input signal which determines whether the machine can fetch an instruction. With the use of this signal, **SMA** can be flushed, whereas we have no way of flushing **MA**.

Proof of SMA. The proof of SMA given in [18, 19] uses a variant of the BD notion of correctness. The main theorem proved is that if the SMA starts in SMA_0 , a flushed state, takes n steps to arrive at state SMA_n , also a flushed state, then there is some number m such that stepping the projection of SMA_0 m steps results in the projection of SMA_n . The projection of an SMA state is the ISA state obtained from the program counter, register file, and memory of the SMA state. Notice that for any system which never reaches a flushed state, the above condition is trivially true. Since MA is such a system, proving that it satisfies this condition is of no use.

The proof of SMA also includes a “liveness” component: it is proved that any SMA state can be flushed. Taking these two conditions together, we can ask: are there any pipelined machines for which we can prove the above theorems, but which are obviously not correct? The answer is yes and using Sawada’s proof scripts, we provide a mechanically checked proof that a pipelined machine which does nothing satisfies this notion of correctness. This proof and any other mechanical proof that we claim to have completed can be found on the author’s Web page [20].

Flushing Proof of MA. Even though there is no way to flush MA, we can use flushing to prove that MA is a refinement of ISA. This digression allows us to discuss issues related to refinement maps. One approach is to modify MA so that it can be flushed (which would give us SMA), but this is a different machine. The approach we take is to define an auxiliary function that flushes an MA state. Using this function we show that MA is a refinement of ISA. Notice that in contrast to the proof of SMA, there is no trivial pipelined machine that will satisfy this notion of correctness. This is because proving a WEB between a pipelined machine and ISA implies that any ISA behavior can be matched by the pipelined machine; since ISA has non-trivial behaviors so does the pipelined machine. Even so, this proof is not entirely satisfactory and this has to do with the use of flushing as a refinement map.

When we define systems at different levels of abstraction, we often find that there are inconsistent states, *e.g.*, consider an MA state in which the first latch contains an instruction that is not in memory. This is usually dealt with by considering only “good” (reachable) MA states. The flushing-based refinement imposes no such restriction because pipelined machines are self-stabilizing: from any state they eventually reach a good state and flushing guarantees this. As a result, the refinement map—which is supposed to show us how to view MA states as ISA states—relates inconsistent MA states with consistent ISA states (all ISA states are “good”). We find such a refinement objectionable.

Proof of MA. The approach we take to pipelined machine verification in the rest of this paper is to prove a WEB where the refinement map relates pipelined machine states to the ISA states obtained by retaining the programmer visible components of the committed part of the pipelined state. The definition of the refinement map is based on the function `committed-MA` which takes an MA state

and returns the **MA** state obtained by invalidating all partially completed instructions and moving the program counter back based on the number of partially completed instructions. As mentioned previously, “good” **MA** states are the ones reachable from a committed state. The function **good-MA** recognizes **MA** state s if **committed-MA.s**, stepped the appropriate number of times, is s . Notice that as with flushing, this function is easy to define because we can use the definition of **MA**. In fact, it is simpler to define than the flushing operation because we are not trying to get the machine into a special state: we are just stepping it. Notice that the use of **good-MA** allows us to avoid defining an invariant (an error prone process), hence, we maintain this methodological feature of the **BD** approach. We call this approach the “commitment approach.”

ISA128. An **ISA128** state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an exception flag. The next state of an **ISA128** state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the program counter is incremented and the target register is modified to contain sum , the sum of the values in the source registers if the sum is less than 2^{128} . Otherwise, if an overflow occurs, the exception flag is checked; if it is off, then the program counter is incremented and the target register is assigned $sum \pmod{2^{128}}$; if the exception flag is on, the exception handler is called. The exception handler is a constrained function of the program counter, register file, and memory that returns a new program counter, register file, memory, and exception flag. (A function about which we know only that it satisfies some specified properties is called a constrained function. An uninterpreted function is a special case of a constrained function.) Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. **ISA128** is the specification for **MA128**, **MA128serial**, and **MA128net**.

MA128, MA128serial, and MA128net. **MA128** is a three-stage pipelined machine. An **MA128** state is a six-tuple consisting of a program counter, a register file, a memory, two latches, and an exception flag. As with **MA**, the three stages are the fetch stage, the set-up stage, and the write-back stage. If an overflow occurs during an arithmetic operation, then the partially executed instructions are invalidated and the interrupt handler is called. The resulting state is constrained to be flushed (*i.e.*, both latches are invalid). We prove that **MA128** is a refinement of **ISA128** using the commitment approach.

MA128serial is the same as **MA128**, except that the ALU is defined in terms of a serial adder and a multiplier based on the adder. The adder, multiplier, and proof of their correctness are taken from [10]. We used the commitment approach to prove that **MA128serial** refines **MA128**. Since the ALU of **MA128** operates on bit-vectors, the refinement map used maps the bit-vectors in the register file and the second latch to numbers. By Theorem 1 (composition), we get that **MA128serial** is a refinement of **ISA128**. Although the use of the

composition theorem here was not essential, it was nice to be able to break up the proof into these two logically separate concerns. ACL2 can take advantage of the structural similarity between **MA128serial** and **MA128**, hence the proof of correctness is pretty fast. This is covered in more detail later.

MA128net is the same as **MA128**, except that the ALU is defined in terms of an adder described in a netlist language. The netlist adder is a 128-bit adder and is described in terms of primitive functions on bits. We have a function that generates an adder of any size and we prove that the adder generated is correct by relating it to the serial adder. We prove that **MA128net** is a refinement of **MA128serial**, hence by composition, a refinement of **ISA128**.

3.3 Non-deterministic Machines

We now consider the non-deterministic versions of the six deterministic machines described above. The names of these machines are: **ISAint**, **MAint**, **ISA128int**, **MA128int**, **MA128intserial**, and **MA128intnet**. They are elaborations of the similarly named deterministic machines, except that they can be interrupted. Whereas the next state of the deterministic machines is a function of the current state (even in the presence of exceptions), the next state of the machines described in this section also depends on the interrupt signal, which is free. Therefore, the machines in this section are non-deterministic.

The approach in [1] to dealing with interrupts is different. There, the correctness criterion is: if M_0 is a flushed state and if taking n steps where the interrupts at each step are specified by the list l results in a flushed state M_n , then there is a number n' and a list l' such that stepping the projection of M_0 n' steps with interrupt list l' results in the projection of M_n . Notice that a machine which always ignores interrupts satisfies this specification.

In our approach, we have to show that the pipelined machine is a refinement of the specification, as before. Note that this is the same final theorem we proved in the deterministic case, as WEBs can be used to relate non-deterministic systems. Therefore, our notion of correctness cannot be satisfied by a pipelined machine that ignores interrupts. Another advantage is that our proof obligation is still about single steps of the machines, as opposed to finite behaviors. The problem with the finite behaviors approach was highlighted above: when comparing finite executions of a pipelined machine and of its specification, there are executions with different lengths; how does one relate interrupts in one execution with interrupts in the other?

ISAint. An **ISAint** state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an interrupt register. The next state of an **ISAint** state is obtained by first checking the interrupt register. If non-empty, the interrupt handler is called. The interrupt handler is a constrained function of the register file, memory, and interrupt register and returns a state with the same pc and register file, but may modify memory, and clears the interrupt register. If the interrupt register is empty, we check if an interrupt has been raised. If so, we

record the interrupt type in the interrupt register. If not, we proceed by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the target register is modified to contain the sum of the values in the source registers and the program counter is incremented. Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. **ISAint** is the specification for **MAint**.

MAint. **MAint** is a three-stage pipelined machine. An **MAint** state is a six-tuple consisting of a pc, a register file, a memory, two latches, and an interrupt register. The three stages are the fetch stage, the set-up stage, and the write-back stage, as before. The next state of an **MAint** state is obtained by first checking the interrupt register. If non-empty, partially executed instructions are aborted and the interrupt handler is called. Otherwise we check if an interrupt has been raised, in which case we abort partially executed instructions and set the interrupt register. Otherwise, execution proceeds in a fashion similar to the execution of **MA**. The refinement map used to show that **MAint** is a refinement of **ISAint** is almost identical to the one used to show that **MA** is a refinement of **ISA**, except that the interrupt register is also retained.

ISA128int. As the name implies this is the ISA-level specification of 128-bit ALU machine with exceptions and interrupts. Interrupts are given priority, and this machine is defined the way you would expect: it is similar to **ISAint**, except that arithmetic operations are checked for overflows, in which case the exception handler is called. This machine is the specification used for the machines **MA128int**, **MA128intserial**, and **MA128intnet**.

MA128int, MA128intserial, and MA128intnet. **MA128int**, **MA128intserial**, and **MA128intnet** are three-stage pipelined machines analogous to **MA128**, **MA128serial**, and **MA128net**, respectively, but with exceptions. As before, we show that **MA128int** is a refinement of **ISA128int**, that **MA128intserial** is a refinement of **MA128int** (where the refinement map converts bit-vectors to integers) and finally that **MA128intnet** is a refinement of **MA128intserial**. By the composition theorem, we get that all of these machines are refinements of **ISA128int**.

4 Proof Decomposition

To reduce the amount of guidance that has to be manually given to the theorem prover, we develop a methodology with mechanical support which we use to verify the various variants of Sawada's machine. We outline the approach in this section by discussing the proof that **MA** is a refinement of **ISA**.

A user of the ACL2 theorem prover often uses books provided by others. Books are files of ACL2 definitions and theorems, *e.g.*, there are books for reasoning about arithmetic, data structures, floating-point arithmetic, and the like.

Books can be certified and then included (*i.e.*, loaded) in future sessions without having to re-admit the definitions and theorems. Books are also used to structure proofs, by placing related definitions and theorems in the same book. The proof that **MA** is a refinement of **ISA** uses some general-purpose books, *e.g.*, a book about arithmetic and books that we have developed for proving that a concrete system is a refinement of an abstract system.

The books specific to the proof are "**ISA**", "**MA**", and "**MA-ISA**". "**ISA**" and "**MA**" contain the definitions of **ISA-step** and **MA-step**, the functions to step (*i.e.*, compute the next state of) the **ISA** and **MA** machines, respectively. The "**MA-ISA**" book contains the definition of the refinement map, **MA-to-ISA**, and the function recognizing "good" **MA** states, **good-MA**. These functions are described in Section 1. Also included is the definition of a rank function, **MA-rank**. The rank function is very simple: it is either 0, 1, or 2 based on how many steps it will take **MA** to commit an instruction. The rest of "**MA-ISA**" consists of three macro calls. These macros are defined in the books we developed for reasoning about **WEBs**. Only these definitions are required; the proof obligation is generated by the macros and discharged by **ACL2**. No user provided theorems, intermediate abstractions, or invariant proofs are needed. The books containing the verification of the other machines are similar. Some of them require auxiliary lemmas, but the lemmas are about the serial adder, the netlist description language, and the like.

We now explain what the three macro calls mentioned above do. The first, **generate-full-system**, is given seven arguments. They are the names of the functions to step and recognize **ISA** and **MA** states, **MA-to-ISA**, **good-MA**, and **MA-rank**. This macro, as the name implies, generates the system to be verified. Recall that in order to prove a **WEB**, we need one system; this system is the union of the **MA** and **ISA** systems. The function **R**, corresponding to the transition relation (\rightarrow in the definition of **WEBs**) of the system is defined. **R** is a function of two arguments that holds between **ISA** states and their successors and **MA** states and their successors. The function **wf-rel** is defined; this is a function of two arguments that holds if the first argument is an **ISA** state, the second argument is a good **MA** state and **MA-to-ISA** of the **MA** state equals the **ISA** state. **B** is defined to be the reflexive, symmetric, transitive closure of **wf-rel** and corresponds to the equivalence relation in the definition of **WEBs**. **Rank** is defined to return 0 on **ISA** states and **MA-rank** otherwise and corresponds to the rank function in the definition of **WEBs**. The function **take-appropriate-step** is defined to return **ISA-step** of its argument, if an **ISA** state, or **MA-step** of its argument otherwise. Several other functions corresponding to existentially quantified formulae are also defined.

The second macro called is **prove-web** and it is given five arguments. They are the names of the functions to step and recognize **ISA** and **MA** states and **MA-rank**. This macro generates the following proof obligation:

```

(defthm b-is-a-wf-bisim-core
  (let ((u (ISA-step s))
        (v (MA-step w)))
    (implies (and (wf-rel s w)
                  (not (wf-rel u v)))
              (and (wf-rel s v)
                    (e0-ord-< (MA-rank v) (MA-rank w))))))

```

Most of ACL2's effort goes into establishing the above theorem. This theorem says that if u is the **ISA-step** of s , v is the **MA-step** of w , and s and w are related by **wf-rel**, but u and v are not, then **wf-rel** relates s and v and the **MA-rank** of v is less than that of w . **E0-ord-<** is the less than relation on ACL2 ordinals. (The proofs in this paper depend only on the natural numbers, an initial segment of the ordinals.) Compare this theorem with the definition of WEBs. We used domain-specific information to remove the quantifiers and much of the case analysis. For example, in the definition of WEBs, u ranges over successors of s and v is existentially quantified over successors of w , but because we are dealing with deterministic systems, u and v are defined to be *the* successors of s and w , respectively. Also, **wf-rel** is not an equivalence relation as it is neither reflexive, symmetric, nor transitive, but **wf-rel** relates **ISA** and **MA** states and this is enough for us to automatically deduce the theorem for the induced equivalence relation. Finally, we ignore the second disjunct in the third condition of the definition of WEBs because **ISA** does not stutter. The use of this domain-specific information makes a big difference, *e.g.*, when we tried to prove the theorem obtained by a naive translation of the WEB definition, ACL2 ran out of memory after 30 hours, yet the above theorem is now proved in about 11 seconds. **Prove-web** also generates some “type” theorems that we do not discuss further.

The final macro called is **wrap-it-up** and it is given the same seven arguments given to the first macro call. What this macro does is a little bit more interesting. It generates the proof obligations required to show the WEB by generating the following three theorems (compare with the definition of WEBs):

```

(defequiv B)

(defthm rank-well-founded
  (e0-ordinalp (rank x)))

(defthm b-is-a-wf-bisim
  (implies (and (B s w)
                (R s u))
            (or (exists-w-succ-for-u w u)
                (and (B u w)
                     (e0-ord-< (rank u) (rank s)))
                (exists-w-succ-for-s w s))))

```

That is, B is an equivalence relation, rank is a well-founded witness, and the third condition in the definition of WEBs. (EO-ordinalp is a predicate that recognizes ACL2 ordinals and recall that R is the transition relation.) What is interesting is how the final theorem is proved. It is proved using a proof rule of ACL2 called *functional instantiation* (see [14]). In one of the books that support reasoning about WEBs, we proved that from a theorem similar to $\text{b-is-a-wf-bisim-core}$, a theorem similar to b-is-a-wf-bisim follows. The theorem proved was about a constrained system, *i.e.*, a system about which we have some constraints, but not a definition. Using functional instantiation, we can prove b-is-a-wf-bisim by showing that the system defined by MA and ISA satisfies the constraints of the constrained system. In this way, we prove a decomposition theorem once about a constrained system that allows us to reduce our problem to one with no quantifiers and minimal case analysis, and we can use the decomposition on any system that satisfies the constraints.

We end this section by describing a clear, compositional path from the verification of term-level descriptions of pipelined machines to the verification of low-level descriptions (*e.g.*, netlist descriptions). In our proof that MA128 is a refinement of ISA128, the definition of the ALU is “disabled”; the result is that ACL2 treats the ALU as an uninterpreted function. However, to verify that MA128net is a refinement of MA128, we “enable” the definition of the ALU and prove theorems relating a netlist description of the circuit to a serial adder which is then related to addition on integers. This allows us to relate term-level descriptions of machines to lower-level descriptions in a compositional way.

5 Conclusions

Various approaches to pipelined machine verification appear in the literature. In [1], a notion of correctness based on flushing and commuting diagrams is presented. Another approach using skewed abstraction functions is outlined in [22, 3, 23]. There are approaches based on model-checking, *e.g.*, in [24], compositional model checking and symmetry reductions are used and in [2] assume-guarantee reasoning at different time scales is discussed. There are theorem-proving based approaches, *e.g.*, in [25, 26, 27, 28], an intermediate abstraction called MAETT is used to verify some very complicated machines. Another theorem-proving approach is presented in [8, 4, 29], where “completion functions” are used to decompose the abstraction function. In [20], a related notion of correctness based on state and temporal abstraction is used to verify a pipelined machine. In addition, there is work on decision procedures, *e.g.*, [2, 23] present decision procedures for boolean logic with equality and uninterpreted function symbols and in [5] their decision procedure is used to verify pipelined machines.

We have presented an approach to pipelined machine correctness based on WEBs and argued that only machines which truly implement the instruction set architecture satisfy our notion of correctness. In contrast, we showed with mechanical proof that the Burch-Dill variant of correctness used in [20, 23] can be satisfied by trivial, clearly wrong, implementations. We verified various ex-

tensions to the simple pipelined machine presented in [12, 13]. Our extensions include exception handling, interrupts, and ALUs described in part at the netlist level. In every case, we proved the same final theorem. In addition, we showed how to use the compositionality of WEBs and ACL2's functional instantiation to relate term-level descriptions to netlist-level descriptions. All of the proofs were done within one logical system and we thus avoided the semantic gaps that could otherwise result. To automate the proofs, we developed a methodology with mechanical support that is used to generate and discharge the proof obligations. The main proof obligation generated contains no quantifiers and has minimal case analysis, but once proved is used to automatically infer the WEB. This is done with the functional instantiation of a decomposition theorem that is part of the mechanical support for WEBs that we provide. All of the proofs are available from the author's Web page [14].

Using our methodology and notion of correctness, we were able to prove Sawada's example ("Proof of MA" in Section 3.1) without the use of any intermediate abstractions or invariants. The size of the file containing the proof, which does not include the machine definitions, is about 3K; the size of the files containing Sawada's proof is about 94K. The time required for our proof (including the loading of related books) is about 30 seconds on a 600MHz Pentium III; the time required for Sawada's proof is about 460 seconds (on the same machine).

For future work, we plan to look at more complicated machines, namely machines with deeper pipelines, out-of-order execution, and richer instruction sets.

Acknowledgements

J Moore has been a great resource and inspiration. He also provided a proof of the equivalence between the serial adder and the netlist adder. Rob Sumners and Kedar Namjoshi read the paper and made many useful suggestions.

References

- [1] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.
- [2] M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59, 1988.
- [3] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification—CAV '99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.
- [4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

- [5] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI, Dec. 1993.
- [6] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification-CAV '99*, volume 1633 of *LNCS*, pages 208–221. Springer-Verlag, 1999.
- [7] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 8–22. Springer-Verlag, 1999.
- [8] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of a pipeline microprocessors. In A. J. Hu and M. Vardi, editors, *Computer-Aided Verification-CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
- [9] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification-CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.
- [10] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, June 2000.
- [11] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, July 2000.
- [12] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL <http://www.cs.utexas.edu/users/moore/acl2/>.
- [13] M. Kaufmann and J. S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 2000. To appear, See URL <http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html#foundations>.
- [14] P. Manolios. Homepage of Panagiotis Manolios, 2000. See URL <http://www.cs.utexas.edu/users/pete/>.
- [15] P. Manolios. Well-founded equivalence bisimulation. Technical report, Department of Computer Sciences, University of Texas at Austin, 2000. In preparation.
- [16] P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification-CAV '99*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.
- [17] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.
- [18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.
- [19] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.
- [20] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.
- [21] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification-CAV '99*, volume 1633 of *LNCS*, pages 455–469. Springer-Verlag, 1999.

- [22] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999. See URL <http://www.cs.utexas.edu/users/sawada/dissertation>.
- [23] J. Sawada. Verification of a simple pipelined machine model. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 137–150. Kluwer Academic Press, 2000.
- [24] J. Sawada and W. A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.
- [25] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.
- [26] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, Sept. 1990.
- [27] M. K. Srivas and S. P. Miller. Formal verification of an avionics microprocessor. Technical Report CSL-95-04, SRI International, 1995.
- [28] P. J. Windley and M. L. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design*, volume 901 of *LNCS*, pages 33–52. Springer-Verlag, 1994.

Do You Trust Your Model Checker?*

Wolfgang Reif¹, Jürgen Ruf², Gerhard Schellhorn¹, and Tobias Vollmer¹

¹ Universität Augsburg, Lehrstuhl für Softwaretechnik und Programmiersprachen
D-86135 Augsburg, Germany

{reif,schellhorn,vollmer}@informatik.uni-augsburg.de

² Wilhelm-Schickard-Institute, University of Tübingen, D-72076 Tübingen, Germany
ruf@informatik.uni-tuebingen.de

Abstract. In this paper we describe the formal specification and verification of the efficient algorithm for real-time model checking implemented in the model checker RAVEN. It was specified and proved using the KIV system. We demonstrate how to decompose the correctness proof into several independent subtasks and indicate the corresponding verification efforts. The formal verification revealed some errors, reduced the code size, and improved the efficiency of the implementation.

1 Introduction

Model checking is an important technique to detect errors or to prove their absence in safety critical soft- and hardware systems. Model checking automatically verifies properties of state based systems. For efficiency, it is usually implemented using highly optimized data structures and algorithms. On the other hand, when a property can be shown, the only result we usually get from a model checker, is a “yes”. The absence of a comprehensible proof raises the question: can the model checker be trusted?

In this paper, we will answer this question for the case of the real-time model checker RAVEN [1]. RAVEN uses time-extended finite state machines (interval structures) to describe systems and a timed version of CTL (clocked CTL, CCTL) to describe their properties. Optimized algorithms based on extended characteristic functions are used to compute the extension sets in the model checker.

Our solution consists in the application of formal methods to ensure the correctness of formal methods: We apply the interactive specification and verification system KIV to formalize and prove the algorithms of RAVEN. To our knowledge, this is the first case study tackling formal verification of a state-of-the-art real-time model checker. This paper is the result of the cooperation of two groups, in the context of a research program on formal methods for engineering applications*: the developer of RAVEN [2, 3] (second author), and the development group of KIV [4] (remaining authors).

* This work is supported by the DFG (Deutsche Forschungsgemeinschaft) under the priority program “Integrating Software Specification Techniques for Engineering Applications”

The KIV case study described in this paper consists of four steps: first, we define a formal specification of the semantics of CCTL, the basic model checking algorithm and the optimizations. Second, we verify the correctness of the simple and the optimized algorithm. Third, we give an efficient implementation of the abstract algorithms based on bitvectors, and finally we prove the implementation correct. This implementation relies on a standard software package for extended characteristic functions [1] (based on multiterminal BDDs, MTBDDs [2]). We formalized the interface to this package in KIV. The verification relies on the specification of this interface. Verifying [1] against the interface is an independent subtask which was not part of the case study.

Our case study shows that it is possible to give a modular specification, such that the correctness of the model checker can be split into several independent verification tasks. With the help of the correctness proofs we found some critical definition errors in the formal specification of the optimizations. After correcting them we proved that the basic algorithms and the optimizations using bitvectors are correctly implemented. Parts of the code were shrunk. One prediction function (see Sect. 3.1) worked too pessimistic and was optimized.

In Section 4, we will describe interval structures and the logic CCTL, which constitutes the basis of RAVEN. Section 5 discusses the used optimizations and the efficient implementation. Section 6 gives our approach to formalization and verification. An overview over the specifications and correctness proofs is presented in Section 7. Section 8 concludes the paper.

2 Real-Time Model Checking

Model checking is a well established method for the automatic verification of finite state systems. It checks if a given state transition system satisfies a given property specified as a propositional temporal logic formula.

The approach we will examine is developed for timed systems and timed specifications. It is presented in [3]. In this section, we will explain the main ideas behind the model checking verification technique which are necessary for the remaining part of the paper. First we will present the formal model and the temporal logic. Afterwards we will introduce the representation with extended characteristic functions and the main model checking procedure.

2.1 Interval Structures

Interval structures are finite state transition systems. The transitions are labeled with intervals of natural numbers to represent delay times. The structures use the notion of clocks to represent time: every structure contains exactly one clock working in a discrete time domain. A transition is enabled if the actual clock value is within the interval of an outgoing transition. The successor state as well as the delay time is chosen indeterministically w.r.t. the transition relation and the labeled delay intervals. The clock is reset if a transition fires.

Definition 1. An interval structure $\mathcal{J} = (P, S, S_0, T, L)$ is a tuple with a set of atomic propositions P , a set of states S , a set of initial states S_0 , a function $T : S \times S \rightarrow \wp^\omega(\mathbb{N}_0)$ that connects states with labeled transitions and a state labeling function $L : S \rightarrow \wp(P)$.

Every state of an interval structure must be left after the maximal state time: $MaxTime(s) := \max\{v \mid \exists s'. v \in T(s, s')\}$

Besides the states, we now also have to consider the currently elapsed time to determine the transition behavior of the system. Hence, the actual configuration of a system is given by an interval structure state $s \in S$ and the actual clock value $v \in \mathbb{N}_0$. The set of all configurations of an interval structure is given by: $G_{\mathcal{J}} = \{(s, v) \mid s \in S \wedge v \leq MaxTime(s)\}$

The semantics of interval structures is defined over runs. A run is a sequence of configurations $r = (r_0, r_1, \dots)$ with $r_i = (s_i, v_i) \in G_{\mathcal{J}}$ and for all $i \geq 0$ holds either

- $r_{i+1} = (s_i, v_i + 1)$ and $v_i < MaxTime(s_i)$ or
- $r_{i+1} = (s_{i+1}, 0)$ and $v_i \in T(s_i, s_{i+1})$

In the following, we call (s_i, v_i) the local predecessor of $(s_i, v_i + 1)$, which corresponds to the first case of the definition. Similar, we call (s_i, v_i) a global predecessor of $(s_{i+1}, 0)$ if $v_i \in T(s_i, s_{i+1})$. Note that for a set of configurations, only the computation of global predecessors depends on the transition relation T .

2.2 CCTL

CCTL (Clocked Computation Tree Logic) is a propositional temporal logic using quantitative time bounds for expressing real time properties (e.g. bounded liveness). The following definition describes the syntax of CCTL formulas.

Definition 2. Given a set of atomic propositions P . The set of CCTL formulas F_{CCTL} is defined to be the smallest set with

- $P \subseteq F_{CCTL}$
 - if $m \in \mathbb{N}_0, n \in \mathbb{N}_0 \cup \{\infty\}, m \leq n$ and $\varphi, \psi \in F_{CCTL}$ then
 - $\neg \varphi, \varphi \wedge \psi,$
 - $EX_{[m]} \varphi$ (Next), $EF_{[m,n]} \varphi$ (Eventually), $EG_{[m,n]} \varphi$ (Globally),
 - $E(\varphi \cup_{[m,n]} \psi)$ (Until), $E(\varphi S_{[m]} \psi)$ (Successor), $E(\varphi C_{[m]} \psi)$ (Conditional)
 - $AX_{[m]} \varphi, AF_{[m,n]} \varphi, AG_{[m,n]} \varphi, A(\varphi \cup_{[m,n]} \psi), A(\varphi S_{[m]} \psi), A(\varphi C_{[m]} \psi) \in F_{CCTL}$
- The symbol ∞ is defined through: $\forall i \in \mathbb{N}_0 : i < \infty$.

All interval operators can also be accompanied by a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the EX-operator has no time bound, it is implicitly set to one.

For this paper we will only define the semantics for the EF-operator (“Eventually”), the semantics for the other operators may be found in [14]. The semantics of CCTL is given by a model relation (\models):

Definition 3. Given the interval structure $\mathcal{J} = (P, S, S_0, T, L)$, a starting configuration $r_0 \in G_{\mathcal{J}}$ and the CCTL formula $\varphi \in F_{CCTL}$.

$$\mathcal{J}, r_0 \models \text{EF}_{[m,n]}\varphi :\Leftrightarrow \text{there exists a run } r = (r_0, \dots) \\ \text{and an } i \in [m, n] \text{ such that } \mathcal{J}, r_i \models \varphi$$

A formula φ is valid in a model \mathcal{J} , iff $\mathcal{J}, (s, 0) \models \varphi$ for all initial states $s \in S_0$. The defined interval operator may be expressed by operators only carrying an upper time bound, e.g. $\text{EF}_{[m,n]}\varphi := \text{EX}_{[m]}\text{EF}_{[n-m]}\varphi$.

2.3 The Basic Model Checking Algorithm

The main idea of model checking algorithms is the following: building the syntax graph of the formula to check, computing bottom-up sets of configurations representing sub formulas (called extension sets), checking if the set of initial states is a subset of the extension set of the complete formula.

The computation of the extension sets is done by a function *ext*. The extension sets of atomic formulas (i.e. the leaves of the syntax tree) are given by the labeling function L and the possible clock values in the appropriate states of the interval structure \mathcal{J} . Boolean connections can be computed by applying the corresponding set operations on the extension sets. Finally, the computations of the extension sets of temporal logic operators are defined recursively. We will present them using the EF-operator as an example:

$$\begin{aligned} \text{ext}(\text{EF}_{[0]}\varphi) &:= \text{ext}(\varphi) \\ \text{ext}(\text{EF}_{[n+1]}\varphi) &:= \text{ext}(\varphi) \cup \text{ext}(\text{EX}(\text{EF}_{[n]}\varphi)) \end{aligned}$$

The operator EX (“Next”) states, that a certain formula is fulfilled after the next step. The time bound n for $\text{EF}_{[n]}$ has to be finite. For $n = \infty$ we can show, that the extension sets reach a fixpoint, i.e. $\exists i. \text{ext}(\text{EF}_{[i]}) = \text{ext}(\text{EF}_{[i+1]})$. This means, that the recursion can be terminated if the $\text{ext}(\text{EF}_{[i]})$ will not change anymore. The realization of the recursive definition of the EF-operator leads to the algorithm shown in Fig. 1.

All other CCTL operators can be implemented with similar algorithms. The operators EX, ES, EC, which do not reach a fixpoint during computation cannot be computed for $n = \infty$.

Two questions remain: how are the sets of configurations represented in order to achieve efficient computations and how is the EX-operator computed?

A main advance in the field of model checking was made with the introduction of symbolic representations of state spaces [4]. Instead of an explicit enumeration of sets of states, they are represented by characteristic functions. For our time extended model checking algorithm, we use an extension of characteristic functions (ECF) which maps interval structure states to sets of associated clock values [14]. The function $A_C : S \rightarrow \wp(\mathbb{N})$ represents a set $C \subseteq G_{\mathcal{J}}$ of configurations:

$$A_C(s) := \{ v \mid (s, v) \in C \}$$

```

confset EF(confset C, natinfty n, transrel T)
begin
  confset old :=  $\emptyset$ , R := C;
  while n > 0  $\wedge$  old  $\neq$  R do
    old := R;
    R := C  $\cup$  EX(R,1,T);
    CORRECTED 30.6.2000 n := n - 1;
  end
  return R;
end

```

Fig. 1. The basic EF-algorithm

In the following, we will use sets of configurations and the ECFs representing these sets synonymously, i.e. we will write C instead of Λ_C . Since we aim at verifying an implementation of a model checker we will furthermore use the notation and intuition of MTBDDs (multi-terminal BDDs [1, 2]), which are used to implement extended characteristic functions.

MTBDDs representing sets of configurations code the state space in the decision diagram and associate a set of natural numbers representing the clock values with each state (i.e. each leaf of the decision diagram). An example MTBDD representing the configuration set $\{(a, 3), (a, 4), (b, 5), (ab, 2)\}$ is depicted in the dashed box in Fig. 1. Set operations can be implemented by performing the appropriate operations on the leaves of the MTBDD. The transition relation is represented similarly by two cascaded MTBDDs (cf. Fig. 2). The first one represents the state space of the model \mathcal{J} , the second represents the sets of predecessor configurations of the individual states. This corresponds to the fact that often predecessors, but never successors are computed during model checking. The dashed box in Fig. 2 represents all predecessor configurations of state b .

The second open question is, how can the extension set of an EX-operator be computed? If the extension set of its argument formula is known, the extension set of the EX-operator is given by all predecessor configurations consisting of the union of local and global predecessors.

The local predecessors of a set of configurations may be computed by a function *local-pre* as follows. Using MTBDDs, the computation may be reduced to the leaves of the MTBDD where each clock value contained is decremented.

Global predecessors only exist for configurations with a zero clock value. The computation is done by a function *global-pre*(C, T). For every state s' with $(s', 0) \in C$, the set of predecessor configurations is looked up in the transition relation and the resulting configuration sets are combined to a new configuration set.

Putting these considerations together, $\text{EX}_{[1]}$ can be computed as:

$$\text{EX}(C, 1, T) = \text{local-pre}(C) \cup \text{global-pre}(C, T)$$

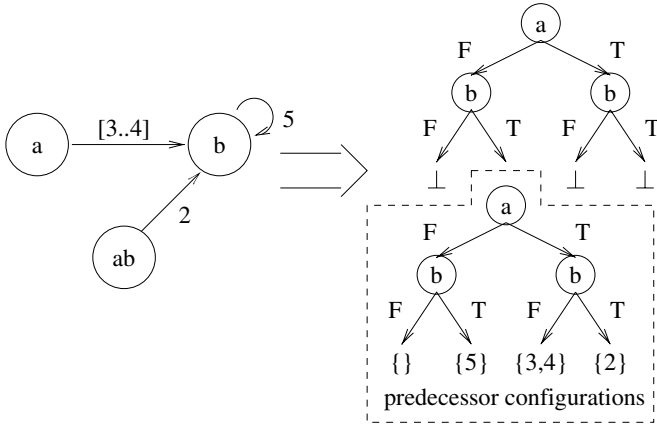


Fig. 2. Representation of the transition relation using two cascaded MTBDDs

3 The Central Idea of the RAVEN Model Checker

3.1 Time Prediction

If we analyze the predecessor computation we observe the following:

- The computation of global predecessors is more expensive than the computation of local predecessors, since the complete transition relation must be inspected.
- Often, the global predecessors do not change between two predecessor computations.

We use a technique called time prediction to overcome the single step traversal and to avoid unnecessary global predecessor computations [11, 12]. The idea is to define a time prediction function that computes how many steps the global predecessors stay constant.

Again, we exemplarily discuss the EF-operator. Although the basic idea of time prediction can be applied to all operators, every temporal operator needs a separate prediction function.

The time prediction function *predict-EF* is computed locally, i.e. for each state separately by a function *local-pr-EF*. The minimum of the prediction times *mp* is the time span which can elapse without any change in the set of global predecessors. Arguments of *local-pr-EF* are the sets of clock values $c \subseteq \mathbb{N}_0$ and $g \subseteq \mathbb{N}_0$ which contain the last interim result of the computation and the results of the last computation of global predecessors:

$$\text{predict-EF}(C, G) = \min_{s \in S} \text{local-pr-EF}(C(s), G(s))$$

$$\text{local-pr-EF}(c, g) := \begin{cases} v & \text{if } v = \min(c, g - 1) \wedge v > 0 \\ \infty & \text{otherwise} \end{cases}$$

The set operation $g - 1$ decrements all members of g by one.

After the prediction the fixpoint iteration of the temporal operators may be performed mp times. Analogous to the above, a function *apply-EF* is defined which performs the fixpoint iteration locally for every state by using the recursively defined function *local-EF*.

$$\text{apply-EF}(C, G, mp)(s) = \text{local-EF}(C(s), G(s), mp)$$

$$\text{local-EF}(c, g, mp) = \begin{cases} c & \text{if } mp = 0 \\ \text{local-EF}(c, g, mp - 1) \cup \text{local-EF}(c, g, mp - 1) - 1 \cup g & \text{otherwise} \end{cases}$$

Putting together the above definitions and considerations, we obtain the optimized algorithm shown in Fig. 1.

```

confset EF'(confset C, natinfy n, transrel T)
begin
  confset old :=  $\emptyset$ , R := C, G :=  $\emptyset$ ;
  natinfy p;
  while n > 0  $\wedge$  old  $\neq$  R do
    old := R;
    G := global-pre(R, T);
    p := predict-EF(R, G);
    if p > n then p := n;
    R := apply-EF(R, G, p);
    n := n - p;
  end
  return R;
end

```

Fig. 3. The optimized EF-algorithm

3.2 Time Jumps Using Bitvectors

The local fixpoint iteration needs $O(mp)$ set operations for execution. We define a technique called time jumping which replaces this iterative execution by an efficient implementation using either bitvectors or interval lists.

Using interval lists has the advantage of little memory consumption but lacks the efficiency of the bitvector based algorithms. Hence, we now will concentrate on the bitvector based implementation.

The implementation *local-EF#* for the EF-operator using bitvectors is shown in Fig. 2. Bitvectors are used to represent sets of natural numbers. A number n is contained in a CORRECTED 30.6.2000 set, iff the n th bit of the bitvector representation is 1. The basic idea of the implementation of *local-EF#* is to traverse bitvectors: in the n th step of the algorithm the n th bit of the input and

an internal state of the algorithm are used to compute the n th bit of the result. Hence, we only need a single specialized operation to compute *local-EF* instead of $O(mp)$ set operations when using the recursive definition.

4 Formal Specification and Verification Concept

Our case study consists of five parts:

1. Specification of CTL and its semantics. The left half of Fig. 1 illustrates this step.
2. Specification of the basic model checking algorithm and verification, that the algorithm implements the semantics (cf. right half of Fig. 1).
3. Specification of the optimized algorithms with time prediction for the temporal operators (e.g. EF' as defined in Fig. 1) and verification that they yield the same results as the simple recursive version (e.g. EF as given in Fig. 1). Figure 1 visualizes this step.
4. Nonrecursive definition *local-EF'* of the local computations *local-EF* used in EF' and verification, that both definitions are equivalent (cf. upper half of Fig. 1).
5. Implementation of *local-EF'* based on bitvectors and correctness proof for the implementation. The lower half of Fig. 1 shows this part.

These five parts will be discussed in the five subsections of the next section. All five parts were specified using the structured, algebraic specifications of KIV, which are similar to the standard algebraic specification language CASL [1]. To do the correctness proofs, KIV offers a concept of modules, which describe a refinement relation between specifications. KIV automatically generates proof obligations that assure the correctness of modules.

An important goal in the design of the case study was to structure specifications, such that each of the four verification steps could be expressed as the correctness of one module. This has two advantages: First, each module can be verified independently of all others, which means that the correctness problem is decomposed into several orthogonal subproblems. Second, a general theory (see [1]) of modular systems (consisting of specifications and modules) assures, that a correct model checking algorithm (that implements the \models predicate) can be “plugged” together by repeatedly replacing abstract definitions with more concrete versions: Starting with the unoptimized algorithm *modelcheck*, first EF is replaced with EF' (and similar for the other temporal operators), then the call to *local-EF* in EF' is replaced with a call to *local-EF'*, and finally this call is replaced again with the bitvector implementation. The final algorithm is identical to the one used in RAVEN, except that it has an abstract interface of extended characteristic functions. Their implementation using actual MTBDD operations could be plugged in using another module (which could be separately verified).

Developing a modular system of specifications and modules, such that “plugging the algorithm together” and separate verification of the steps described

above became possible, was a major creative step in this case study. Two important design decisions were to use higher-order operations on ECFs (*apply* and *reduce*, see Sect. 3.4) to have an abstract interface to BDDs, and to use the intermediate nonrecursive definition *local-EF'* (see Sect. 3.4).

Technically, modular systems are developed in KIV using a graphical representation, called development graphs. Such a graph contains rectangular boxes for specifications and rhombic boxes for modules. Arrows represent the structure of specifications and implementation relations. The following section will show for each of the five steps some relevant part of the development graph and sketch the contents of the specifications. Putting all parts together, i.e. merging the development graphs of figures 4, 5 and 6 gives the full modular specification and implementation of the model checker. Full details on all specifications, modules and proofs can be found in [10].

5 Verification of Correctness

5.1 Specification of CCTL Semantics

The structure of the algebraic specification for CCTL and its semantics is shown in the left half of Fig. 7. The main predicate specified in the top-level specification is $\mathcal{J}, (s, v) \models \varphi$ (φ holds in configuration (s, v) over model $\mathcal{J} = (T, L)$). Two typical axioms of this specification are

$$\begin{aligned} \mathcal{J}, (s, v) &\models \text{EF}_{[n]} \varphi \\ \Leftrightarrow \exists r. \text{run}(r, T) \wedge \text{first}(r) = (s, v) \wedge \exists i. i \leq n \wedge \mathcal{J}, r_i &\models \varphi \\ 30.6.2000 (T, L), (s, v) &\models p \Leftrightarrow p \in L(s) \end{aligned}$$

The definition is based on a specification of the data type of CCTL formulas, the specification *transrel* of the transition relation of an interval structure and (indirectly) on the specification *confsets* of configuration sets (in algebraic terms, the top-level specification is an enrichment of *transrel* and *confsets*). The transition relation is defined as a function $T : \text{state} \rightarrow (\text{state} \rightarrow \text{set}(\text{nat}))$. $T(s')(s)$ gives the possible delay times for a transition from state s to state s' . Configuration sets are specified as functions $C : \text{state} \rightarrow \text{set}(\text{nat})$. A configuration set C contains a configuration (s, v) , iff $v \in C(s)$. This representation corresponds to extended characteristic functions.

Both configuration sets and the transition relation are specified as actualizations of a generic datatype of extended characteristic functions $ecf : \text{state} \rightarrow \text{elem}$: the parameter type *elem* is instantiated by *set(nat)* and *confset* respectively. The use of generic ECFs allows us to be fully abstract in our correctness analysis of the model checking algorithms, while it is still possible to implement ECFs with MTBDDs (with *elem* being the type of the BDD leaves) and to verify this implementation separately.

¹ The set P of atomic propositions, and the set S of states are carrier sets in our algebraic specification. Therefore they need not be explicitly mentioned in the definition of a model

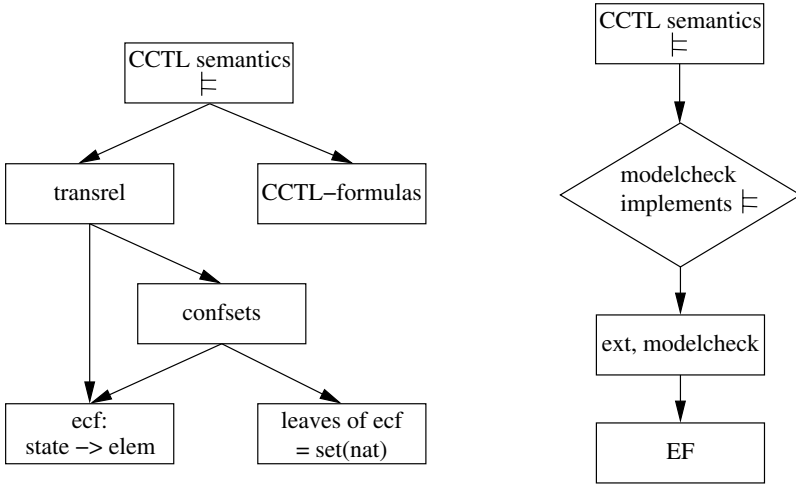


Fig. 4. Specification of CCTL Semantics

5.2 Correctness of Simple Model Checking

The right half of Fig. 1 shows the part of the KIV development graph that deals with the correctness proof of the simple model checking algorithm with respect to the semantics of CCTL (the structure of the specification of the CCTL semantics above the module has now been omitted).

Specification of Simple Model Checking. The specification shown below the module in Fig. 1 contains the simple model check algorithm *modelcheck*. It is specified by the following axioms:

$$\begin{aligned}
 \text{modelcheck}(\mathcal{J}, c, \varphi) &\leftrightarrow c \in \text{ext}(\varphi, \mathcal{J}), \\
 \text{ext}(\text{EF}_{[n]} \varphi, (T, L)) &= \text{EF}(\text{ext}(\varphi), n, T), \\
 \text{ext}(\neg \varphi, \mathcal{J}) &= G_{\mathcal{J}} \setminus \text{ext}(\varphi, \mathcal{J}), \\
 (s, v) \in \text{ext}(p, (T, L)) &\leftrightarrow p \in L(s) \wedge (s, v) \in G_{(T, L)}, \\
 &\dots
 \end{aligned}$$

Again, the specification is based on configuration sets and the transition relation (not shown in the figure). It is also based on a subspecification which defines a tail-recursive function *EF*, which computes the extension set of the temporal operator $\text{EF}_{[n]}$. The specification also contains similar functions for the other temporal operators, but like in the previous sections, we will now concentrate on the implementation of *EF*. We have preferred the tail-recursive version over the program given in Fig. 1 for two reasons: First, the specification remains independent of an implementation language. Second, proofs using the tail-recursive function are smaller compared to proofs using while-loops and invariants.

To define the computation of local and global predecessors, two generic higher-order operations *apply* and *reduce* on ECFs are used:

$$\begin{aligned} \text{local-pre}(C) &= \text{apply}(C, -1) \\ \text{global-pre}(C, T) &= \text{reduce}(T, (\lambda C_0, s'. \text{ if } 0 \in C(s') \text{ then } C_0 \text{ else } \emptyset), \cup, \emptyset) \end{aligned}$$

apply(*ecf*, *f*) applies a function *f* on each leaf of *ecf*. *reduce*(*ecf*, *f*, *f'*, *a*) applies the function *f* on each leaf of *ecf* and combines the results using function *f'*, starting with the value *a*. The function -1 used in the definition of *local-pre* decrements each number contained in a leaf of an ECF and drops zeros. The λ expression contained in the definition of *global-pre* looks up the predecessors of all states that contain a zero clock value.

Proof of Correctness. The KIV module shown in the development graph of Fig. ■ automatically generates proof obligations. We must show, that the simple model checking algorithm *modelcheck* satisfies all axioms of the predicate \models , i.e. that *modelcheck* implements the predicate \models .

$$\mathcal{J}, c \models \varphi \leftrightarrow \text{modelcheck}(\mathcal{J}, c, \varphi)$$

Proving these proof obligations is straightforward using theorems that assure the existence of fixpoints for the operators EF, EG, etc. and takes only a few hours of verification time.

etc. theory for bounded,

5.3 Time Jumps and Time Prediction

Figure ■ shows the part of the development graph that is relevant for the verification of the optimization step that introduces time prediction and time jumps.

Specification of Time Prediction. Most parts of the specification of the optimized version *EF'* of the computation (cf. Fig. ■) can be adopted from the unoptimized algorithm. The functions required to specify time prediction and time jumps, *predict-EF* and *apply-EF* are defined using the functions *apply* and *reduce*:

$$\begin{aligned} \text{apply-EF}(\text{ecf}) &= \text{apply}(\text{ecf}, \text{local-EF}) \\ \text{predict-EF}(\text{ecf}) &= \text{reduce}(\text{ecf}, \text{local-pr-EF}, \text{min}, \infty) \end{aligned}$$

Thus, the functions are reduced to functions *local-EF* and *local-pr-EF* which work on the leaves of the ECFs. The specification of the latter functions follows directly the definition in Sect. ■.

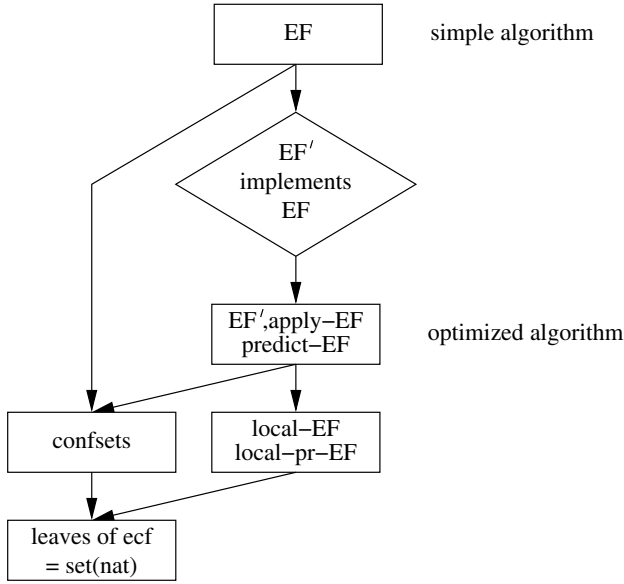


Fig. 5. Specification of time-prediction

Proof of Correctness. To prove correctness of the module, it must be shown, that all axioms of the simple algorithm are also satisfied by the optimized algorithm. The main theorem needed in the proof is:

$$\begin{aligned}
 & n < \text{predict-EF}(C, \text{global-pre}(C, T)) \\
 \rightarrow & \text{global-pre}(C, T) \\
 = & \text{global-pre}(\text{apply-EF}(C, \text{global-pre}(C, T), n), T)
 \end{aligned} \tag{1}$$

It expresses the central idea of the optimization step, that the global predecessors do not change during the computation of as many steps as the time prediction permits. Since *predict-EF* yields the minimum of the results of the predicted values for all leaves, the proof can be done by reducing the theorem to the leaves of the ECF considered and proving the analogous theorem for each single leaf:

$$\begin{aligned}
 & p = \text{local-pr-EF}(C(s), G(s)) \wedge (n < p \vee p = \infty) \\
 \rightarrow & (0 \in \text{local-EF}(C(s), G(s), n) \leftrightarrow 0 \in C(s))
 \end{aligned} \tag{2}$$

Here, the term $0 \in \text{local-EF}() \leftrightarrow \dots$ states that the global predecessors of the leaf considered remain unchanged. For reasons explained in the next section, we assume property (2) as an axiom here and postpone its proof until then.

The proof obligation for the operator *EF* is proved by induction over the number of steps the operator computes. Two important theorems are needed in the induction step of the proof.

The first,

$$\text{EF}'(C, n + 1, T) = \text{EF}'(\text{EF}'(C, n, T), 1, T) \tag{3}$$

ensures, that a single step can be split off from the computation of EF' . The proof is conducted by induction over n . Expanding the definition of a single recursion “computes” p steps of the operator using function $apply\text{-}EF$. Since only one step has to be split off, a similar decomposition lemma is also needed for $apply\text{-}EF$. This lemma can be shown by proving the following, analogous lemma for the leaves of the ECF:

$$\text{local-EF}(c, g, n + 1) = \text{local-EF}(\text{local-EF}(c, g, n), g, 1) \quad (4)$$

The second important theorem is required, because the recursion schemes of the simple and of the optimized version are slightly different. While the simple recursion $EF(C, n + 1, T) = C \cup EX(EF(C, n, T), 1, T)$ always adds its argument C to the interim result, the optimized version calls $apply\text{-}EF(R, p)$ with the interim result R to compute p steps. Hence, in any step of the algorithm, the result of the last major step is added to the configuration set. Since the operator considered increases monotonically, both recursion schemes produce the same results:

$$\begin{aligned} R &= EF'(C, n, T) \\ \rightarrow R \cup EX(R, 1, T) &= C \cup EX(R, 1, T) \end{aligned} \quad (5)$$

Results of Verification. During verification, we found erroneous time prediction functions for two operators, *local-pr-EU* (for strong-until) and *local-pr-ES* (for the successor operator). In some cases, the original definition of *local-pr-EU*,

$$\text{local-pr-EU}(c_1, c_2, g) = \begin{cases} n & \text{if } n = \min(c_1, g + 1) \wedge [0, n] \subseteq c_2 \\ \infty & \text{otherwise} \end{cases}$$

yields too high values. Therefore the algorithm sometimes forgets to recompute the global predecessors, which leads to incorrect results. The corrected version of *local-pr-EU* (corrections **bold**) is

$$\text{local-pr-EU}(c_1, c_2, g) = \begin{cases} n & \text{if } \neg \mathbf{0} \in \mathbf{c}_1 \wedge n = \min(c_1, g + 1) \\ & \wedge [0, n - \mathbf{1}] \subseteq c_2 \\ \infty & \text{otherwise} \end{cases}$$

Inspection of both implementations of RAVEN showed, that the bitvector based version behaves correctly. The implementation using interval lists contained the error and was subsequently corrected.

The time prediction *local-pr-ES* not only contained a case where too high results were computed, but also a too pessimistic case. Since the definition of *local-pr-ES* was rather complex (8 lines), we corrected it by reduction to the EX operator. This led to a much more compact definition using only 3 lines.

The verification also showed a critical point in the computation of $EX_{[n]} \varphi$ (next operator), that was not detected during design and informal analysis of the

algorithm. Normally, the **EX** operator, which does just computations of predecessors, never reaches a fixpoint. Nevertheless, cycles (i.e. $\text{EX}_{[m]} \varphi = \text{EX}_{[m+p]} \varphi$) may occur in the computation. To stop the computation as early as possible, the simple version of **EX** (which is similar to the algorithm in Fig. 4) performs a fixpoint test after every step of the computation by comparing $\text{EX}_{[m]} \varphi$ to $\text{EX}_{[m+1]} \varphi$. The optimized version contains this fixpoint test, too. But since the optimized version computes p steps at a time, $\text{EX}_{[m]} \varphi$ is compared to $\text{EX}_{[m+p]} \varphi$. Therefore, if the computation of $\text{EX}_{[n]} \varphi$ with $n > m + p$ contained such a cycle of length p , the computation would stop too early.

However, we could show, that this situation can never occur, because the time-prediction function permits either an infinite number of steps or only one step at a time if a cyclic computation takes place:

$$\begin{aligned} \text{apply-EX}(C, G, n) &= C \wedge p = \text{predict-EX}(C, G) \wedge n \leq p \\ \rightarrow p &= 1 \vee p = \infty \end{aligned} \quad (6)$$

Verification Effort. All seven temporal operators implemented in RAVEN were proven correct. We found, that the proofs of all operators share a common pattern. This pattern consists of theorems (4), (5), (6), (7), (8) and some auxiliary theorems. Additionally, for operators, that do not reach a fixpoint during computation, a theorem like (9) was needed. Due to this pattern, the verification time required decreased from one week for the first operator to about 2 days. Although the complexity of the operators increased, the correctness proofs all have about the same length, because the growing experience helped to compensate the extra complexity with higher automation.

5.4 Nonrecursive Representation of Time Jumps

A look at the implementation of time jumps (cf. Fig. 4) and time prediction shows, that the computations of these programs do not fit to the recursive definition of *local-EF* very well. Therefore, even simple proofs using the recursive definition are technically very complex. Therefore we decided to introduce a nonrecursive function *local-EF'*, which describes the results of the operator **EF**:

$$\begin{aligned} &\text{local-EF}'(c, g, n) \\ &= \{v \mid \exists v_0 \in c \cap [v, \dots, v + n]\} \\ &\quad \cup \{v \mid \exists v_0 \in g \cap [v, \dots, v + n - 1] \wedge n \neq 0\} \end{aligned} \quad (7)$$

Since we want to use this function instead of the recursive function *local-EF*, we first have to prove the equivalence of both representations. Again, this is done using a KIV module. The corresponding part of the development graph is depicted in the upper half of Fig. 5. The proof obligations generated for the module ensure, that the nonrecursive function *local-EF'* satisfies the axioms of *local-EF*.

An additional advantage of this approach is, that we can use the nonrecursive function *local-EF'* to prove the theorems which use *local-EF* and *local-pr-EF*. To

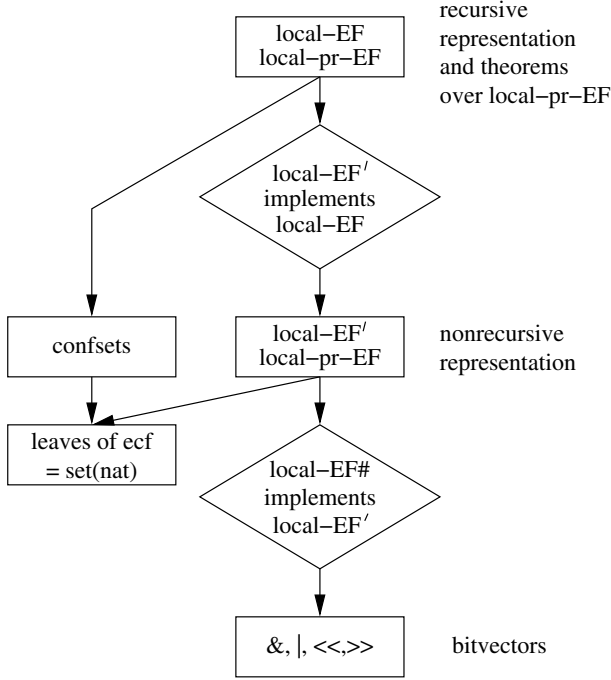


Fig. 6. Specification of explicit representation of time jumps

do this, we added these theorems as axioms in the specification which contains *local-EF* and used them as assumption in the previous section.

Now, we get these theorems as additional proof obligations for the nonrecursive definition *local-EF'* in the module.

The proofs of these theorems do not require any new ideas – they are typical for proofs over sets of natural numbers. Therefore, a discussion is omitted. The time required to do the correctness proofs was about two days per operator, including the proofs of the theorems assumed in the previous verification step.

5.5 Implementation Using Bitvectors

The previous sections were concerned with deriving an efficient model checking algorithm on abstract data types. This section considers the efficient implementation of the optimized algorithm. RAVEN offers two representations. One that represents sets of natural numbers with bitvectors and one that uses interval lists. Since some of the algorithms used in the latter representation were already verified in an earlier case study with KIV [14] (unrelated to this project), and bitvectors are used as the default in RAVEN, we decided to verify this version.

The implementation verified was derived directly from the C++ source-code of RAVEN by omitting code concerned with memory allocation and partitioning of bitvectors into words.

Again, the verification was done using a KIV module, which is shown in the lower half of Fig. 7. In contrast to the previous sections, real programs are used in the module. The programs *local-EF#* and *local-pr-EF#* depicted in Fig. 7 implement the functionality of *local-EF'* and *local-pr-EF* using bitvectors.

Bitvectors are defined to be strings of binary digits without leading zeros. In addition, the basic operations $\&$ (binary and), \mid (binary or), \ll (logical shift left), \gg (logical shift right), a length function $\#$ and a bit-selection function denoted by array-like subscripts are defined.

<pre> bitvec local-EF#(c, g, n) var m = 0, r = 0, pos = #(c g) + 1, state = 0 in r := 0; if n = ∞ then m := #(c g) + 1 else m := n; while pos \neq 0 do pos := pos - 1; if g[pos] = 1 then state := m; if c[pos] = 1 then state := m + 1; if state \geq 1 then r := (1 \ll pos) r; state := state - 1; end end return r end </pre>	<pre> nat local-pr-EF#(c, g) var n = 0 in if c = 0 \wedge g = 0 then n := ∞ else if c[0] = 1 then n := ∞ else var pos = 0 in n := 0; while n = 0 do if c[pos] = 1 then n := pos; else if g[pos] = 1 then n := pos + 1; pos := pos + 1; end end return n end </pre>
---	--

Fig. 7. Implementation of time-jump function *local-EF*

As proof obligations of the module it must be shown, that the implementation programs terminate and satisfy the axioms of *local-EF'* and *local-pr-EF* (for a general introduction to the theory of program modules see [4]; verification techniques for proof obligations in KIV are discussed in [1]).

To stay as close as possible to the implementation of RAVEN we decided to consider bitvectors without leading zeros only. This restriction is formalized as a predicate *r*. Additional proof obligations are generated by the KIV system to ensure that the programs terminate and keep the restriction invariant.

The correctness proofs for *local-EF#* and *local-pr-EF#* both require invariants for the while-loops. The one for *local-pr-EF#* was easy to obtain, since the current state of computation depends on few factors only. The invariant for *local-EF#* is shown in Fig. 8. It consists of two major parts. *INV₁* states, that the postcondition is satisfied for the computations made so far. The main difficulty of the proof was to construct *INV₂*. It describes the variable *state*, which represents the “memory” of the algorithm. The construction of the invariants took several iterations and days, depending on the complexity of the operator and

$$\begin{aligned}
\text{INV}_{\text{EF}} &\equiv \text{INV}_1 \wedge \text{INV}_2 \\
\text{INV}_1 &\equiv \forall n_1. \text{pos} \leq n_1 \\
&\quad \rightarrow (\quad \text{r}[n_1] = 1 \\
&\quad \leftrightarrow (\exists i. \text{c}[i] = 1 \wedge \neg i < n_1 \wedge \neg n_1 + n < i) \\
&\quad \vee (\exists i. \text{g}[i] = 1 \wedge \neg i < n_1 \wedge \neg n_1 + n - 1 < i) \wedge n \neq 0) \\
\text{INV}_2 &\equiv \text{r}(r) \wedge \text{state} \leq n \\
&\quad \wedge (\quad \text{state} = 0 \vee \text{c}[\text{pos} + n - \text{state}] = 1 \\
&\quad \vee \text{state} < n \wedge \text{g}[\text{pos} + n - \text{state} + 1] = 1) \\
&\quad \wedge (\forall i. i < n - \text{state} \rightarrow \text{c}[\text{pos} + i] = 0) \\
&\quad \wedge (\text{state} < n \rightarrow (\forall i. i < n - \text{state} + 1 \rightarrow \text{g}[\text{pos} + i] = 0))
\end{aligned}$$

Fig. 8. Invariant for while-loop of *local-EF* procedure

the number of “memory” variables. The size of the invariants ranges between 11 and 25 lines. Once the correct invariant was found, the proofs were large, but easy and automatic.

Summarizing, the effort taken to prove the correctness of the bitvector based implementation of RAVEN was about 2 weeks. On average, it took three iterations to find the correct invariant. A beneficial side effect of the verification was the discovery of inefficient and redundant code. The implementation of *local-EU* could be shortened from 73 to 18 lines of code.

6 Conclusion

In this paper we investigated the correctness of an optimized real-time model checking algorithm. We demonstrated that it is possible to develop the efficient implementation from the specification of the semantics in a series of refinement steps. Each CORRECTED 30.6.2000 step could be verified independently of the others.

During the verification, several errors were discovered in the time prediction functions, which constitute the heart of the optimization. Also, some inefficient code could be eliminated. We were able to define a common scheme for the correctness proofs of the different temporal operators, which reduced the verification effort significantly.

The time required to formalize and verify the model checking algorithm was about 4 months. Compared to the total time that was needed to develop and implement the optimizations, the extra effort was modest.

With the verification of the kernel algorithm we now feel justified to answer the question posed in the title of this paper with “yes”, assuming the standard BDD package works correctly. Since model checkers are often used in safety critical applications, we hope our results encourage further research in their correctness.

References

1. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 188–191, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.
3. E. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J.C.-Y. Yang. Spectral Transforms for large Boolean Functions with Application to Technologie Mapping. In *ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, Dallas, TX, June 1993.
4. CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. <http://www.brics.dk/Projects/CoFI>
5. D. Long. Long-package sun release 4.1 overview of c-library functions, 1993.
6. W. Reif. Correctness of Generic Modules. In Nerode and Taitlin, editors, *Symposium on Logical Foundations of Computer Science*, LNCS 620, Berlin, 1992. Logic at Tver, Tver, Russia, Springer.
7. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.
8. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer Academic Publishers, Dordrecht, 1998.
9. J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.
10. Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.
11. Jürgen Ruf and Thomas Kropf. Using MTBDDs for Composition and Model Checking of Real-Time Systems. In *FMCAD 1998*. Springer, November 1998.
12. Jürgen Ruf and Thomas Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. IFIP WG 10.5, Springer, September 1999.
13. T. Vollmer. Korrekte Modellprüfung von Realzeitsystemen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).

Executable Protocol Specification in ESL[★]

E. Clarke¹, S. German², Y. Lu¹, H. Veith^{1,3}, and D. Wang¹

¹ Carnegie Mellon University

² IBM T. J. Watson Research Center

³ TU Vienna

Abstract. Hardware specifications in English are frequently ambiguous and often self-contradictory. We propose a new logic ESL which facilitates formal specification of hardware protocols. Our logic is closely related to LTL but can express all regular safety properties. We have developed a protocol synthesis methodology which generates Mealy machines from ESL specifications. The Mealy machines can be automatically translated into executable code either in Verilog or SMV. Our methodology exploits the observation that protocols are naturally composed of many semantically distinct components. This structure is reflected in the syntax of ESL specifications. We use a modified LTL tableau construction to build a Mealy machine for each component. The Mealy machines are connected together in a Verilog or SMV framework. In many cases this makes it possible to circumvent the state explosion problem during code generation and to identify conflicts between components during simulation or model checking. We have implemented a tool based on the logic and used it to specify and verify a significant part of the PCI bus protocol.

1 Introduction

Motivation. The verification of bus protocols, i.e., of communication protocols between hardware devices, as in the case of the well-known PCI bus, is a central problem in hardware verification. Although bus protocol design and verification have become increasingly important due to the integration of diverse components in IP Core-based designs, even standard bus protocols are usually described in English which makes specifications often ambiguous, contradictory, and certainly non-executable.

Example 1. It is often the case that English documentation attaches different meanings to a single name in different situations, as in the following definition for “data phase completion” from PCI 2.2 [15], page 10: *A data phase is completed on any clock both IRDY# and TRDY# are asserted.* On Page 27, however, there is a different definition: *A data phase completes when IRDY# and [TRDY# or STOP#] are asserted.* The obvious problem with these definitions is whether data phase completion should include the case when both IRDY# and STOP# are asserted.

[★] This research is sponsored by the Gigascale Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of GSRC, NSF, or the United States Government.

By carefully reading the English documentation, one can also find many contradictory statements, as in the following example from PCI 2.2 [15], page 50: *Once the master has detected the missing DEVSEL#, FRAME# is deasserted and IRDY# is deasserted.* On Page 51, however, it is said that *Once a master has asserted IRDY#, it can not change IRDY# or FRAME# until the current data phase completes.* The reason why these two are contradictory is that the first sentence allows FRAME# to be deasserted without the current data phase being complete, while the second one disallows this.

Traditional hardware description languages are usually not well-suited for protocol specification because they are based on existing concrete designs (or abstractions thereof) instead of specifications, and their execution model therefore focuses on single-cycle transitions. With protocols, the specification is naturally represented by constraints on signals which may connect relatively distant time points.

Another problem of transition-system based approaches is that naive composition of participants in the protocol may cover up important protocol inconsistencies due to synchronization faults or write conflicts among non-cooperative participants. It is important that the specification language be executable, i.e., that a machine model can be computed from the specification. This is a trivial property for hardware description languages, but not for protocol specifications.

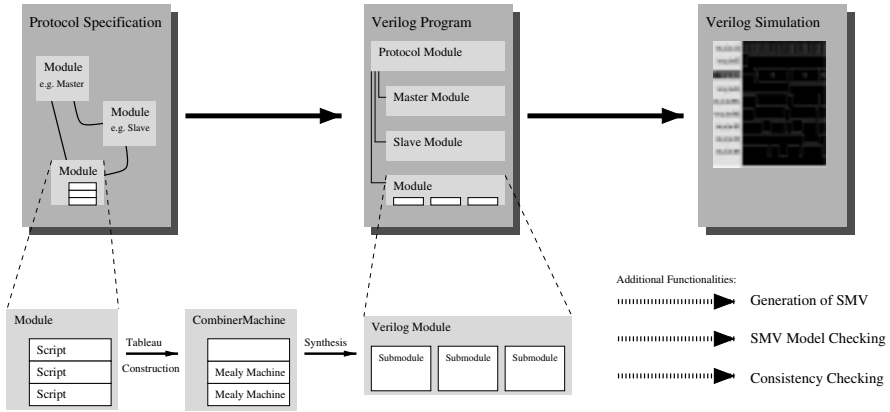


Fig. 1. System Overview

The ESL Logic. We propose the new logical language **ESL** which facilitates specification, verification, and simulation of protocols. ESL is based on a variant of linear temporal logic (LTL) where atomic propositions are constraints on signals in the protocol, e.g. $REQ \in \{0, 2\}$. The core of ESL are *ESL scripts*, i.e., finite collections of executable temporal formulas. More precisely, a script is a finite collection of axioms $\Phi_P \Rightarrow \Phi_X$ where Φ_P is a temporal formula involving only the temporal operator **P** (immediate past-time), and Φ_X is a temporal formula involving only the temporal operator **X**, e.g. $P a \Rightarrow b \vee X X a$. Atoms in Φ_P are interpreted as tests of previous

signals, while Φ_X asserts constraints in the future. Scripts may contain local variables which are not visible in the traces (i.e., models) of the script.

An *ESL module* \mathcal{E} is a finite collection $\mathcal{S}_1 \parallel \dots \parallel \mathcal{S}_n$ of *ESL scripts*. The intended semantics is that ESL scripts describe Mealy Machines and that the ESL module specifies the synchronous composition of the Mealy machines. Different scripts of a module potentially conflict if they employ common output signals. Note that in general two scripts $\mathcal{S}_1 \parallel \mathcal{S}_2$ describe two different Mealy machines, while the conjunction $\mathcal{S}_1 \wedge \mathcal{S}_2$ of two scripts is itself a script which describes a single Mealy machine.

A finite collection \mathcal{P} of ESL modules with pairwise disjoint output signals is called an *ESL protocol*. The semantics of an ESL protocol is the synchronous composition of its modules. ESL modules cannot directly conflict with each other, but they may have receptiveness failures (see Section 4).

In real protocols, ESL scripts describe distinct functionalities and features of hardware devices, while ESL modules correspond to the devices themselves. Note that different scripts can constrain the same signal, which is important when translating natural language specifications.

Experiments and Results. We show that semantically ESL expresses exactly the regular safety properties. This property is crucial for synthesizing executable models of the protocol, and distinguishes it from other executable logics, e.g. [10]. A fragment of LUSTRE has been shown to capture the same expressive power [12] in the context of synchronous data-flow languages for real-time systems. We present efficient tableau and synthesis algorithms to obtain executable models of hardware protocols, cf. Figure 1. The tricky aspect of synthesizing hardware protocols is to group the signals occurring in the specification into output signals and input signals. To this aim, we introduce a marked tableau construction which preserves the distinction between past observations and future assertions. Since each script is converted into a Mealy machine separately, there is no state explosion during the translation phase.

The execution of ESL in Verilog and SMV makes it possible to use the power of well-known and highly developed verification paradigms; in particular, it is possible to verify important features about the ESL protocol, such as synchronization contradictions between different scripts and receptiveness [2] of the ESL modules.

Another important debugging method is property checking, i.e., existing Verilog monitors and model checking tools can be easily used to debug ESL protocols. To improve coverage of the Verilog simulation, a dynamically biased random simulation test bench [12] can also be written directly in ESL scripts.

In a case study, we have used ESL to specify the PCI bus protocol Rev 2.2 [15]. Several errors were identified, including some errors from the English specification. Verilog monitors and temporal formulas from [12] have been checked against the generated Verilog and SMV models.

Related Work. Algorithms for synthesizing concurrent and distributed programs from temporal logic specifications have been developed for CTL by Clarke and Emerson [13] and for LTL by Manna and Wolper [14]. Both methods synthesize global state transition systems based on an interleaved asynchronous model of computation and then use projection to obtain controllers for individual processes. Neither of these methods has been

very successful because of the computational complexity of the decision procedures involved. Protocol synthesis, especially synthesis of bus protocols, is easier, since many implementation details are already given. Our methodology automatically exploits this information by incorporating it into the synthesis process.

Various formalisms have been used in hardware specification and synthesis [1, 2, 3, 4]. Recently, Shen and Arvind used term rewriting systems to specify and verify ISA and out-of-order implementations of processors [5].

Gabbay has developed an executable logic [6] for AI applications in which programs are written using rules of the form “If A holds in the past then do B”. Since liveness properties can be expressed in his logic, it is not suitable for Verilog simulation.

Originally, our work was motivated by the approach of Shimizu, Dill and Hu [7]. They use monitors to specify bus protocols formally, based on rules of the form “predicate on past states \Rightarrow predicate on current state”. A coding style is defined that promotes readability, discourages errors, and guarantees receptiveness. We use a similar definition of receptiveness (see Section 5). Although our paper is also concerned with formal specification of bus protocols, there are several important distinctions between the two papers. First, our formalization is a *declarative formalism* based on temporal logic. Second, their monitors are restricted to interface signals. Consequently, constructing appropriate monitors can be tricky. For example, identification of “master abort” in the PCI bus protocol involves observing the bus for several cycles. Third, both Verilog and SMV models can be obtained from our specifications. Our Verilog model can directly generate valid simulation patterns and can, therefore, be simulated with part of the real design. In the monitor approach, the Verilog versions of the monitors are only applicable to a complete design, because monitors can not be used to generate valid patterns.

Consistency problems arise at different levels of specifications. Bryant et al [8] have developed a notion of consistency by identifying combinational dependencies. They show how to derive the consistency model from a modular specification where individual modules are specified as Kripke structures and give an algorithm to check the system for consistency. Very recently, the system FoCs has been developed at IBM Haifa [9]. FoCs automatically generates simulation checkers from RCTL specifications.

A more detailed exposition of the related work and proofs will be given in the full version of this paper.

Structure of the Paper. In Section 2, we describe the logical framework of ESL. Section 3 contains the protocol description language which is used as input to our tool. In Section 4, the translation algorithms are presented. Section 5 shows how to debug and verify protocols. Finally, Section 6 contains our practical experiments with the PCI bus protocol. In Section 7, we briefly summarize our work, and outline future work.

2 The ESL Logic

In this section we describe the linear time logic which underlies the implementation of the ESL system. A more user-friendly input language for ESL will be described in the following section.

2.1 Temporal Logic on Signal Traces

Let $V = \{v_1, \dots, v_n\}$ be a set of variables (signals) where each v_i has finite domain D_{v_i} . A variable v_i is Boolean, if its domain D_{v_i} is $\{0, 1\}$. An *atom* is an expression $v_i = d$, where $d \in D_{v_i}$. The finite set of atoms is denoted by **Atoms**. If v_i is Boolean, then v_i abbreviates $v_i = 1$. *Literals* are atoms and negated atoms. For a set A of atoms, $\text{var}(A) = \{v : \text{for some value } d \in D_i, v = d \in A\}$ is the set of variables appearing in A . A *type* is a consistent conjunction $\bigwedge_{1 \leq i \leq n} \alpha_i$ of literals. As is common in logic, we shall often write types as sets $\{\alpha_1, \dots, \alpha_n\}$. A *complete type* is a type which determines the values of all variables. Note that each complete type can be assumed to contain only (unnegated) atoms.

Example 2. Suppose that $V = \{\text{REQ}, \text{COM}\}$, where $D_{\text{REQ}} = \{1, 2, 3\}$, and $D_{\text{COM}} = \{0, 1\}$. Then **Atoms** = $\{\text{REQ} = 1, \text{REQ} = 2, \text{REQ} = 3, \text{COM} = 0, \text{COM} = 1\}$. Examples of two types σ_1, σ_2 are:

1. $\sigma_1 = \{\text{REQ} = 1, \text{COM} \neq 0\}$, or as a conjunction, σ_1 becomes $\text{REQ} = 1 \wedge \text{COM} \neq 0$.
2. $\sigma_2 = \{\text{REQ} \neq 2, \text{COM} = 1\}$, or as a conjunction $\text{REQ} \neq 2 \wedge \text{COM} = 1$.

σ_1 is a complete type, because it determines the values of both REQ and COM. σ_1 is equivalent to the type $\{\text{REQ} = 1, \text{COM} = 1\}$. σ_2 does not determine the value of REQ, because both $\text{REQ} = 1$ and $\text{REQ} = 3$ are consistent with σ_2 . Therefore, σ_2 is not complete.


ESL is based on a discrete time model, where time points are given by natural numbers from $N = \{1, 2, \dots\}$. A *signal trace* is an infinite sequence $S = s_1 s_2 \dots$, where each s_i determines the values of all variables at time i . S can be viewed as an infinite string, where the alphabet consists of *complete types*. Following the previous example,

$$S = \{\text{REQ} = 1, \text{COM} = 0\} \{\text{REQ} = 1, \text{COM} = 1\} \{\text{REQ} = 3, \text{COM} = 1\} \dots$$

is a signal trace. Thus, the alphabet of signal traces is given by

$$\Sigma = \{\sigma \in 2^{\text{Atoms}} : \sigma \text{ is a complete type without negation}\}.$$

The set of signal traces is given by Σ^ω .

Traces which do not determine the values of all signals are important for us as well (see Section ). For this purpose, we use the alphabet Θ which consists of all types, i.e., *partial assignments* to the signals. Note that Θ is a superset of the signal trace alphabet Σ . Formally, we define

$$\Theta = \{\vartheta : \vartheta \text{ is a type}\}.$$

The set of traces is given by Θ^ω .

Remark. To keep the presentation simple, we tacitly identify two elements of Θ , if they are logically equivalent, e.g., $\text{COM} = 1$ and $\text{COM} \neq 0$. Thus, a rigid formal definition of Θ would require that the alphabet Θ is given by the finite number of equivalence classes of types. For example, we may use the lexicographically minimal types as representatives of their equivalence classes.

Since $\Sigma \subseteq \Theta$, every signal trace is a trace, and all definitions about traces in principle also apply to signal traces. The main difference between Σ and Θ is that each element of Σ determines the values of all signals, while Θ may contain partial information. On the other hand, with each element ϑ of Θ we can associate all elements of Σ which are consistent with ϑ . To this end, we define the function $comp : \Theta \rightarrow 2^\Sigma$ by

$$comp(\vartheta) = \{\sigma \in \Sigma : \sigma \Rightarrow \vartheta\}.$$

$comp$ is called the *completion function*, because it maps a type ϑ to the set of all complete types consistent with ϑ . In other words, $comp$ maps a partial assignment to the set of possible complete assignments. Let $T = t_1 t_2 \cdots \in \Theta^\omega$ be a trace. Then the set of signal traces described by T is given by

$$comp(T) = \{S = s_1 s_2 \cdots \in \Sigma^\omega : \text{for all } i, s_i \in comp(t_i)\}.$$

For a set L of traces $comp(L) = \{comp(T) : T \in L\}$. Given two sets L_1, L_2 of traces, we define $L_1 \approx L_2$ iff $comp(L_1) = comp(L_2)$, i.e., L_1 and L_2 describe the same set of signal traces.

Example 3. Let REQ and COM be as in Example 1. Then the trace $T = (\{\text{COM} = 1, \text{REQ} = 1\} \{\text{COM} = 0, \text{REQ} \neq 2\})^\omega$ does not determine REQ in all positions. It is easy to see that $comp(T)$ is given by the ω -regular expression $(\{\text{COM} = 1, \text{REQ} = 1\} \{\text{COM} = 0, \text{REQ} = 1\} | \{\text{COM} = 1, \text{REQ} = 1\} \{\text{COM} = 0, \text{REQ} = 3\})^\omega$.

A *trace property* L is a set of signal traces. L_n denotes the set of finite prefixes of length n of words in L . A *safety property* is a trace property L , such that for each $t \notin L$, there exists a finite prefix r of t , such that $r \notin L_{|r|}$. In other words, traces not in L are recognized by finite prefixes.

For a set S of atoms and a set $X \subseteq V$ of variables, let S_X denote the restriction of S to atoms containing variables from X only. Similarly, Σ_X denotes the alphabet of complete types for the set of variables X .

V is partitioned into two sets V_G and V_L of *global* and *local* variables. The distinction between global and local variables will be justified in Section 4. Thus, S_{V_G} denotes the restriction of S to *global atoms*, i.e., atoms using global variables. Similarly, Σ_{V_G} denotes the alphabet of complete types for the set of variables V_G . Let **global** be the projection function which maps Σ to Σ_{V_G} by

$$\mathbf{global}(\sigma) = \sigma \cap \mathbf{Atoms}_{V_G}.$$

Intuitively, the function **global** removes all local atoms from the alphabet. With each signal trace $S = s_1 s_2 \cdots$ we associate the global signal trace $\mathbf{global}(S) = \mathbf{global}(s_1) \mathbf{global}(s_2) \cdots$ over $2^{\mathbf{Atoms}_{V_G}}$.

Example 4. Let $S = (\{R = 1, a = 2\} \{R = 0, a = 3\})^\omega$ be a signal trace where a is a local variable. Then $\mathbf{global}(S) = (\{R = 1\} \{R = 0\})^\omega$.

Given two sets L_1, L_2 of traces, we define $L_1 \approx_{\text{gl}} L_2$ iff $\mathbf{global}(comp(L_1)) = \mathbf{global}(comp(L_2))$, i.e., L_1 and L_2 describe the same set of global signal traces.

Lemma 1. *If L is a safety property, then $\text{global}(L)$ is a safety property.*

We consider a linear time logic with temporal operators \mathbf{X} , \mathbf{P} , and \mathbf{G} . Let $T = t_1 t_2 \dots$ be a trace. We inductively define the semantics for atomic formulas f and temporal operators \mathbf{X} , \mathbf{P} , \mathbf{G} as follows:

$$\begin{aligned} T, i &\models f \text{ iff } t_i \Rightarrow f \quad (\text{Here, we use the fact that } t_i \text{ is a type, and thus a formula.}) \\ T, i &\models \mathbf{X} \varphi \text{ iff } T, i + 1 \models \varphi \\ T, i &\models \mathbf{P} \varphi \text{ iff } i \geq 2, \text{ and } T, i - 1 \models \varphi \\ T, i &\models \mathbf{G} \varphi \text{ iff } \forall j \geq i, T, j \models \varphi \end{aligned}$$

The semantics of the Boolean operators is defined as usual. Existential quantification over traces is defined as follows: Let v be a variable, and let $T = t_1 t_2 \dots$ be a trace over $\Sigma_{V - \{v\}}$, i.e., a trace which does not assign values to v . Then we define

$T, i \models \exists v. \varphi$ iff there exists an infinite sequence $a_1 a_2 \dots$ of values for v such that the trace $S = (t_1 \cup \{v = a_1\})(t_2 \cup \{v = a_2\}) \dots$ satisfies φ at time i , i.e., $S, i \models \varphi$. Given a formula φ , $\text{Traces}(\varphi)$ denotes the set of *signal traces* T such that $T, 1 \models \varphi$.

Traces can be combined in a very natural manner:

Definition 1. *Let $T = t_1 t_2 \dots$ and $S = s_1 s_2 \dots$ be traces. Then the combined trace $T \bowtie S$ is given by the infinite sequence $u_1 u_2 \dots$ where*

$$u_i = \begin{cases} t_i \wedge s_i & \text{if } t_i \text{ and } s_i \text{ are consistent} \\ \{\text{false}\} & \text{otherwise} \end{cases}$$

We say that T and S are *compatible* if there is no i such that $u_i = \{\text{false}\}$. Otherwise we say that T and S *contradict*. Let L_1 and L_2 be sets of traces. Then $L_1 \bowtie L_2 = \{T_1 \bowtie T_2 : T_1 \in L_1, T_2 \in L_2\}$.

The above definitions easily generalize to more than two traces. Note that the operation \bowtie introduces the new symbol *false* in the alphabet of traces.

The following important example demonstrates why the operator \bowtie is different from conjunction, and why we need it to analyze traces.

Example 5. Consider the two formulas $\mathbf{G}(a \Rightarrow \mathbf{X} b)$ and $\mathbf{G}(a \Rightarrow \mathbf{X} \neg b)$. Their sets of traces are given by

$$\text{Traces}(\mathbf{G}(a \Rightarrow \mathbf{X} b)) \approx (\{a\}\{b\}|\{\neg a\})^\omega$$

and

$$\text{Traces}(\mathbf{G}(a \Rightarrow \mathbf{X} \neg b)) \approx (\{a\}\{\neg b\}|\{\neg a\})^\omega.$$

When viewed as specifications of different devices, the two formulas are intuitively contradictory when the input signal a becomes true. In fact, in the set of combined traces

$$\text{Traces}(\mathbf{G}(a \Rightarrow \mathbf{X} b)) \bowtie \text{Traces}(\mathbf{G}(a \Rightarrow \mathbf{X} \neg b)) \approx (\{a\}\{\text{false}\}|\{\neg a\})^\omega$$

the contradictions become visible immediately after a becomes true. On the other hand, the naive conjunction $\mathbf{G}(a \Rightarrow \mathbf{X} b) \wedge \mathbf{G}(a \Rightarrow \mathbf{X} \neg b)$ of the formulas is equivalent to $\mathbf{G} \neg a$, their set of traces is given by $\text{Traces}(\mathbf{G} \neg a) = (\{\neg a, b\}|\{\neg a, \neg b\})^\omega$, and thus the potential contradiction vanishes.

2.2 ESL Scripts

The following definition describes the fragment of linear time logic used in ESL.

Definition 2. ESL Scripts

- (i) A *retrospective formula* is a formula which contains no temporal operators except \mathbf{P} .
- (ii) A *prospective formula* is a formula which contains no temporal operators except \mathbf{X} .
- (iii) A *script axiom* Φ is a formula of the form $\Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}$ where $\Phi_{\mathbf{P}}$ is a retrospective formula, and $\Phi_{\mathbf{X}}$ is a prospective formula.
- (iv) An *ESL script* \mathcal{S} is a conjunction $\bigwedge_{1 \leq j \leq k} \Phi_j$ of script axioms Φ_j .
- (v) With each script \mathcal{S} , we associate the formula $\mathbf{G}(\mathcal{S}) = \mathbf{G} \bigwedge_{1 \leq j \leq k} \Phi_j$.

Intuitively, atoms in $\Phi_{\mathbf{P}}$ are interpreted as tests of previous signals, while $\Phi_{\mathbf{X}}$ asserts constraints in the future. For simplicity, we assume that no local variable appears in two different ESL scripts.

Example 6. Consider the ESL script $\mathbf{P} a \wedge a \Rightarrow (\mathbf{X} a \vee \mathbf{X} \mathbf{X} a)$. The script says that if a held true in two subsequent cycles, then a must hold true in one of the two following cycles.

The following lemma says that the temporal operator \mathbf{P} is redundant.

Lemma 2. *Let \mathcal{S} be an ESL script. Then $\mathbf{G}(\mathcal{S})$ is equivalent to a formula of the form $\mathbf{G}(\varphi_{\mathbf{G}}) \wedge \varphi_{\text{Init}}$, where $\varphi_{\mathbf{G}}$ and φ_{Init} are temporal logic formulas which contain no temporal operators except \mathbf{X} .*

The following theorem states that the projection operator **global** achieves the same effect as existential quantification.

Proposition 1. *Let \mathcal{S} be an ESL script, and l_1, \dots, l_n its local variables. Then $\mathbf{global}(\mathbf{Traces}(\mathbf{G}(\mathcal{S}))) = \mathbf{Traces}(\exists l_1 \dots l_n \mathbf{G}(\mathcal{S}))$.*

Thus, projection amounts to a kind of implicit existential quantification. The effect of this quantification is characterized by the following theorem.

Theorem 1. *On global signals, ESL scripts capture the regular safety properties. Formally, for each regular safety property L over Σ_{V_G} , there exists an ESL script \mathcal{S} such that $\mathbf{global}(\mathbf{Traces}(\mathbf{G}(\mathcal{S}))) = L$, and vice versa.*

We conclude from Theorem 1 that projection extends the expressive power of the logic to capture all regular safety properties on *global variables*. We will show in the next section that in praxis the complexity of our logic does not increase significantly. Thus, the addition of local variables appears to be a good choice which balances expressive power and complexity.

Corollary 1. *For global variables, all past time temporal operators, as well as the weak until operator \mathbf{W} are expressible by ESL scripts.*

2.3 Regular Tableaus

The tableau of an LTL formula φ is a labeled generalized Büchi automaton T that accepts exactly the sequences over $(2^{\mathbf{Atoms}_\varphi})^\omega$ that satisfy φ [10]. (Here, \mathbf{Atoms}_φ denotes the set of atomic propositions appearing in φ .) In this section, we define *regular* tableaus by adapting LTL tableaus to ESL.

Definition 3. Regular Tableau

A *regular tableau* \mathcal{T} is a tuple $\langle S^\mathcal{T}, S_0^\mathcal{T}, A^\mathcal{T}, L^\mathcal{T}, R^\mathcal{T} \rangle$ where

- $S^\mathcal{T}$ is a finite set of states, $S_0^\mathcal{T} \subseteq S^\mathcal{T}$ is a set of initial states, and $A^\mathcal{T}$ is a finite set of atoms.
- $L^\mathcal{T} : S^\mathcal{T} \rightarrow \Theta$ is a labeling function which labels states by types, where Θ is the set of types over $A^\mathcal{T}$.
- $R^\mathcal{T} \subseteq S^\mathcal{T} \times S^\mathcal{T}$ is the transition relation of the tableau.

Since by Theorem 1, ESL defines only safety properties, tableaus for ESL do not need all the expressive power of ω -automata. Moreover, the states of regular tableaus are labeled only by sets of atoms (and not by temporal formulas). Intuitively, temporal formulas can be omitted from tableaus because we have local variables which carry the information that is usually carried by the temporal formulas.

Definition 4. Regular Tableau Acceptance

A trace $T = t_1 t_2 \dots \in \Theta^\omega$ is accepted by \mathcal{T} if there exists a sequence $s_1 s_2 \dots \in (S^\mathcal{T})^\omega$ such that

- (i) $s_1 \in S_0^\mathcal{T}$
- (ii) $s_1 s_2 \dots$ is an infinite path in the graph given by the transition relation $R^\mathcal{T}$.
- (iii) For all i , $t_i \Rightarrow L^\mathcal{T}(s_i)$.

The language $\mathcal{L}(\mathcal{T})$ is the set of traces T accepted by tableau \mathcal{T} . Let \mathcal{S} be an ESL script, and \mathcal{T} a tableau. \mathcal{T} is a *correct tableau* for \mathcal{S} , if $\mathcal{L}(\mathcal{T}) \approx \mathbf{Traces}(\mathbf{G}(\mathcal{S}))$, i.e., if the traces generated by the tableau define exactly the signal traces which satisfy \mathcal{S} .

3 ESL Protocols

ESL facilitates modular specification of protocols, in particular hardware protocols. ESL protocols consist of modules which in turn consist of scripts. Under the intended semantics of ESL, modules correspond to distinct devices (e.g. a master and a slave device), while scripts describe independent functionalities of the devices.

Formally, an ESL *module* \mathcal{E} is a finite collection $\mathcal{S}_1, \dots, \mathcal{S}_n$ of scripts. Each script \mathcal{S}_i is given by a finite conjunction $\bigwedge \varphi_j$ of specifications. The intended semantics is that ESL scripts describe Mealy Machines and that the ESL module specifies the synchronous composition of the Mealy machines. Therefore, we shall write $\mathcal{S}_1 \parallel \mathcal{S}_2 \dots \parallel \mathcal{S}_n$ to denote \mathcal{E} . Different scripts for the same module potentially conflict if they employ common output signals, i.e., common global variables which model output signals. Scripts may contain local variables which are not visible to other scripts.

Our aim is to build a machine $M_{\mathcal{E}}$ such that the language accepted by $M_{\mathcal{E}}$ coincides with the combined traces of the scripts on global variables, i.e.,

$$\mathcal{L}(M_{\mathcal{E}}) \approx_{g1} \text{Traces}(\mathcal{S}_1) \bowtie \dots \bowtie \text{Traces}(\mathcal{S}_n).$$

As shown in Example 3.1, trace combination will enable us to identify contradictions between scripts.

A finite collection \mathcal{P} of ESL modules with pairwise disjoint output signals is called an *ESL protocol*. The semantics of protocols is given by the semantics of the *scripts* of its constituent modules. ESL modules of a protocol have fewer sources of conflict among each other than ESL scripts because they do not have common output signals.

3.1 Input Language for ESL Protocols

An ESL protocol must contain exactly one protocol module which starts with the keyword **protocol**. Each module can have three types of variables: **input**, **output** and **local** variables. Each **script** can include more than one specification, which start with the keyword **spec**.

To facilitate easy description of properties and protocols, we augment the basic ESL language with syntactic sugar, such as default values, additional temporal operators, and parametrized macros. In particular, we use the weak until operator **W** and the operator $\text{keep}(a)$ which asserts that the current value of signal a is the same as in the previous step. Note that by Lemma 3.1 we can add all the past temporal operators to the language because they only define regular safety properties. Parameters are used to adapt specifications to hardware configurations of varying size.

Due to space restrictions, we describe the input language by example. The following example is a fragment of the specification of the PCI bus in ESL. It consists of one master and one arbiter.

```

module pci_master
  input      clock, GNT: bool;
  output     REQ = 0: bool;
  local      status : {COMP, MABORT, MTO};
  local      retry_req, last_cycle, addr_phase: bool;
  script     spec  retry_req  $\Rightarrow$  X (retry_req W addr_phase);
              spec  last_cycle  $\Rightarrow$   $\neg$ retry_req;
  script     spec  GNT  $\Rightarrow$  X(keep(REQ)) W status = COMP;
endmodule

module pci_arbiter
  parameter  N;
  input      RST, REQ[1 to N]: bool;
  output     GNT[1 to N]: bool;
  local      tok[1 to N]: bool;
  script spec
    RST  $\wedge$  tok[i]  $\Rightarrow$  X (tok[(i + 1) Mod N] = 1  $\wedge$  tok[i] = 0) for i = 1 to N;
endmodule

```

```

protocol pci_bus
  constant    N = 2;
  modules
    master[1 to N]:      pci_master,
    arbiter(N):          pci_arbiter;
  connection
    master[i].GNT        = arbiter.GNT[i] for i = 1 to N;
    arbiter.REQ[i]       = master[i].REQ[i] for i = 1 to N;
endprotocol

```

This example includes two modules and one protocol module. Each module includes one or two scripts. In the module *pci_master*, REQ has Boolean type and is initialized to 0. The local variable *status* denotes whether the PCI master aborts the transaction or the time out happens. In the module *pci_arbiter*, we define a parameter *N* which will be instantiated when the module gets instantiated. A formula $\varphi(i)$ **for** $i = 1$ **to** N is equivalent to $\varphi(1) \wedge \varphi(2) \wedge \dots \wedge \varphi(N)$. For example, the script of the module *pci_arbiter* is instantiated as a conjunction of two formulas:

$$(RST \wedge token[2] \Rightarrow \mathbf{X} (token[1] := 1 \wedge token[2] := 0)) \wedge (RST \wedge token[1] \Rightarrow \mathbf{X} (token[2] := 1 \wedge token[1] := 0))$$

The protocol module explicitly connects the modules *pci_master* and *pci_arbiter* by matching the corresponding inputs and outputs.

4 Synthesis of Executable Models

Given an ESL protocol, our procedure comprises three main steps to transform executable models into either Verilog or SMV programs.

1. *Preprocessing*. In this step, the ESL protocol is parsed and type checked to make sure each variable is declared and each axiom is well typed. For example, for a Boolean variable x , an atom $x = 3$ is not allowed. Furthermore, the parameters are instantiated and the macros are expanded.
2. *Module Synthesis*. This step converts each ESL module separately into a Verilog or SMV module. An overview of the algorithm **ModuleSynthesis** given in Figure 1. In the algorithm, we first generate a regular tableau for each script and translate the tableau into a nondeterministic automaton. Then we determinize the automaton and translate it into a Mealy machine which can be easily implemented in Verilog or SMV. The combiner connects each Mealy machine. Details of each step will be described in the following three subsections.
3. *Module Connection*. In this final step, the inputs and outputs from different ESL modules are connected. In the synchronous bus protocol design, combinational dependency loops between the signals are not allowed. This step identifies all combinational dependencies within individual modules or between modules. A short outline of this step is given in Section 4.4.

Algorithm ModuleSynthesis($\mathcal{S}_1 \parallel \dots \parallel \mathcal{S}_n$)

```

foreach  $\mathcal{S}_i$ 
   $\mathcal{T}_{\mathcal{S}_i} = \text{MTableau}(\mathcal{S}_i)$ 
   $\mathcal{N}_{\mathcal{S}_i} = \text{Automaton}(\mathcal{T}_{\mathcal{S}_i})$ 
   $\mathcal{N}_{\mathcal{S}_i}^D = \text{Powerset}(\mathcal{N}_{\mathcal{S}_i})$ 
   $M_{\mathcal{S}_i} = \text{GenMealy}(\mathcal{N}_{\mathcal{S}_i}^D)$ 
return Combine( $M_{\mathcal{S}_1}, \dots, M_{\mathcal{S}_n}$ )

```

Fig. 2. Algorithm to synthesize scripts**Algorithm MTableau**(\mathcal{S})

```

mark  $\mathcal{S}$ 
 $\mathcal{T}_{\mathcal{S}} := \text{Tableau}(\mathcal{S})$ 
for all states  $s \in S^{\mathcal{T}_{\mathcal{S}}}$ 
  if clean( $L^{\mathcal{T}_{\mathcal{S}}}$ ) is inconsistent
    then remove  $s$  from  $\mathcal{T}_{\mathcal{S}}$ 
return  $\mathcal{T}_{\mathcal{S}}$ 

```

Fig. 3. Tableau Construction.**4.1 Tableau Construction**

Given an ESL script \mathcal{S} , our aim is to generate a Mealy machine whose operational behavior is specified by the script. Consider the two simple scripts $a = 1 \Rightarrow b = 1$ and $b = 0 \Rightarrow a = 0$, where a and b are Boolean variables. Logically, these scripts are equivalent (since $a \Rightarrow b$ is equivalent to $\neg b \Rightarrow \neg a$.) However, as specifications for Mealy machines they should intuitively describe different machines: the formula $a = 1 \Rightarrow b = 1$ describes a Mealy machine which asserts $b = 1$ if it observes $a = 1$, while $b = 0 \Rightarrow a = 0$ describes a Mealy machine which asserts $a = 0$ if it observes $b = 0$. We conclude from this example that for the operational behavior of the Mealy machines it is important to know for each occurrence of a variable whether it belongs to the retrospective or the prospective part of a script. Variables from the retrospective part eventually will become inputs of Mealy machines, and variables from the prospective part will become outputs of Mealy machines.

In our methodology, we first build a tableau for \mathcal{S} , and then translate it further to a Mealy machine. As argued above, it is important for our tableau construction not to lose the information which variables will be used as outputs of the Mealy machine later on. Therefore, we distinguish such variables by marking them with a symbol \bullet .

Definition 5. Given a set of variables $V = \{v_1, \dots, v_n\}$, V^\bullet is a set $\{v_1^\bullet, \dots, v_n^\bullet\}$ of new variables called *marked* variables.

Given an ESL axiom $\varphi = \Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}$ and the corresponding alphabet Θ , the *marked axiom* φ_m is given by $\Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}^\bullet$ where $\Phi_{\mathbf{X}}^\bullet$ is obtained from $\Phi_{\mathbf{X}}$ by replacing each occurrence of a variable v by the marked variable v^\bullet . ESL scripts are marked by marking all their axioms.

Let X be a set of variables or atoms etc. Then $\text{umk}(X)$ denotes the subset of X where no element contains a marked variable, and $\text{mkd}(X)$ denotes $X - \text{umk}(X)$. The function $\text{clean}(X)$ removes all markers from the elements of X , e.g. $\text{clean}(V^\bullet) = V$.

In Figure 2 we outline the tableau algorithm **MTableau**(\mathcal{S}) for ESL scripts. In the first step, the algorithm marks the script as described above. In the second step we use the standard LTL tableau algorithm described in [14] to generate a tableau for the marked script. Note that from the point of view of the tableau construction algorithm, a variable v and its marked version v^\bullet are different. In the third step, however, we exclude those states from the tableau where the assertions are inconsistent with the observations,

i.e., those states whose labelling would become inconsistent if the marks were removed. Thus, the resulting marked tableau is a correct tableau for \mathcal{S} if the markers are ignored.

Lemma 3. *Let \mathcal{S} be a script. Let L be the set of traces accepted by the tableau $\mathbf{MTableau}(\mathcal{S})$. Then $\text{clean}(L) \approx \text{Traces}(\mathcal{S})$, i.e., after unmarking, the traces of the tableau are the same as the traces of the script.*

Our actual implementation of the algorithm **MTableau** removes inconsistent states on-the-fly *during* constructing tableaux. Thus, fewer intermediate states are created, and less memory is used. In principle, other tableau algorithms [1] could be used, too.

Note that local and global variables are not distinguished by the tableau algorithm. The tableau traces in general contain local variables.

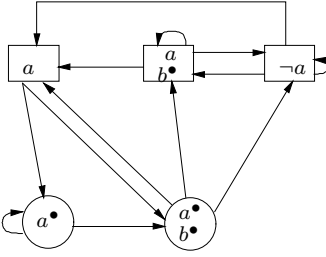


Fig. 4. An example tableau. Boxes denote initial states.

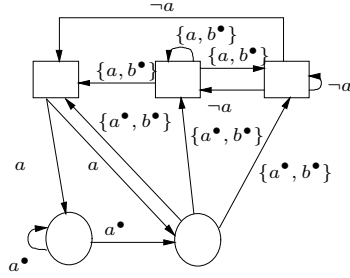


Fig. 5. The corresponding automaton

The following example highlights the construction of tableaux for marked scripts. We will use this example as a running example throughout the paper.

Example 7. Consider the following axiom: $(a \Rightarrow b \vee \mathbf{X} a)$ The meaning of the axiom is: whenever a is asserted, then either b is asserted at the same time point or a is asserted in the next time point. The corresponding marked axiom is: $(a \Rightarrow b^\bullet \vee \mathbf{X} a^\bullet)$ The tableau for this axiom is shown in Figure 4.

Before describing how to translate tableaux into automata, we first give a formal definition of automata.

Definition 6. A *nondeterministic ESL automaton* N is a 4-tuple $\langle S, S_0, \Theta, \delta \rangle$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, Θ is the set of types, and $\delta \subseteq S \times \Theta \times S$ is the transition relation. A trace $T = t_1 t_2 \dots \in \Theta^\omega$ is accepted by N if there exists a sequence $s_1 s_2 \dots \in S^\omega$ such that $s_1 \in S_0$, $s_1 s_2 \dots$ is an infinite sequence of states, and for all i , there exists a type $\vartheta \in \Theta$ such that $t_i \Rightarrow \vartheta$, and $(s_i, \vartheta, s_{i+1}) \in \delta$. The language $\mathcal{L}(N)$ is the set of traces accepted by N .

Given a tableau $\mathcal{T} = \langle S^T, S_0^T, A^T, L^T, R^T \rangle$, the algorithm **Automaton**(\mathcal{T}) computes the automaton $\mathcal{N}_{\mathcal{T}} = \langle S, S_0, \Theta, \delta \rangle$ where $S = S^T$, $S_0 = S_0^T$, and $\delta = \{(s, \vartheta, s') \mid \vartheta \in \Theta, (s, s') \in R^T, L^T(s) = \vartheta\}$. Then the following lemma holds trivially.

Lemma 4. *The tableau \mathcal{T} and the ESL automaton $\mathbf{Automaton}(\mathcal{T})$ accept the same languages. Formally, $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathbf{Automaton}(\mathcal{T}))$.*

4.2 Mealy Machine Synthesis

Since ESL automata are nondeterministic, we cannot translate them directly into Verilog. In this section, we describe how to synthesize automata into deterministic Mealy machines which can then be easily translated into Verilog programs. We proceed in two steps. First, we use a powerset algorithm to determinize the automata. Then, we use the variable markers to determine the inputs and outputs for each state of the Mealy machines.

Since ESL automata do not have Büchi constraints, we use the traditional method for automata determinization by powersets which is described in many textbooks [26].

Although the algorithm **Powerset** is potentially exponential, the resulting automata in our experiments are often much smaller than the original automata. In Figure 4, results for 62 scripts in our PCI protocol specification are shown, where for each script, the ratio of the size of the deterministic automaton compared with that of the original non-deterministic automaton is shown. It can be seen that, in our experiments, the deterministic automata are always smaller than the nondeterministic automata. On average, the automata can be compressed to about 25% of their original size. The intuitive explanation for this behavior is that the powerset algorithm clusters related states into one state. For the example in Figure 5, the automaton after determinization is shown in Figure 7. Since the deterministic automaton accepts the same language as the original

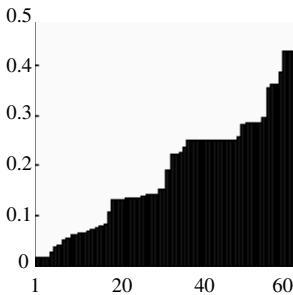


Fig. 6. Compression obtained by determinization

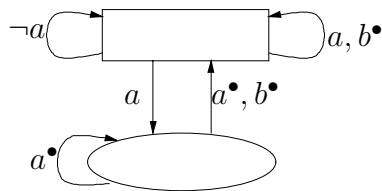


Fig. 7. Automaton after determinization

automaton does, the behavior of the original scripts is maintained after determinization.

Finally, we describe how to generate Mealy machines from deterministic automata. The Mealy machine model presented in the following definition takes into account the type concept which we use for traces.

Definition 7. Mealy Machine

A Mealy machine M is a tuple $\langle S, S_0, I, O, \lambda, R \rangle$ where S is a finite set of states, $S_0 \subseteq$

S is the set of initial states, I is a finite set of input variables, O is a finite set of output variables ($I \cap O = \emptyset$), $\lambda : S \times \Sigma_I \rightarrow \Sigma_O$ is the output function, and $R \subseteq S \times \Sigma_I \times S$ is the transition relation. M is deterministic if for every input σ every state has a unique successor i.e., for all $s, s' \in S$ and $\sigma \in \Sigma_I$, $R(s, \sigma, s') \wedge R(s, \sigma, s'') \rightarrow s' = s''$. M accepts a trace $T = t_1 t_2 \dots$ over alphabet $\Sigma_I \cup \Sigma_O$ if there exists an infinite sequence $S = s_1 s_2 \dots$ of states such that (i) $s_1 \in S_0$; (ii) for all indices i , $R(s_i, \mathbf{umk}(t_i), s_{i+1})$; (iii) for all indices i , $\lambda(s_i, \mathbf{umk}(t_i)) = \mathbf{mkd}(t_i)$. The set of traces accepted by M is denoted by $\mathcal{L}(M)$.

Given a deterministic automaton $N = \langle S, S_0, \Omega, \delta, F \rangle$, we generate a Mealy machine $M = \langle S, S_0, I, O, \lambda, R \rangle$ such that N and M have the same sets of states and initial states. Let MAX be the smallest number such that no state in N has more than MAX immediate successors.

The input variables I of the Mealy Machine M are the unmarked variables of the automaton N , and the new variable nd , $D_{nd} = \{1, \dots, \text{MAX}\}$. Intuitively, the new variable nd will be used to determine the successor state among the MAX possible successor states. The output variables O of M are the marked variables of N , i.e., $O = \mathbf{mkd}(\Omega)$. The output function λ and the transition relation R are defined by the algorithm **GenMealy** shown in Figure 8.

Algorithm GenMealy(N)

```

foreach  $s \in S$ 
     $P = \{\mathbf{umk}(a) \mid \exists s' \in S, \delta(s, a, s')\}$ 
    for each  $a \in P$ 
         $Q = \{b \mid \mathbf{umk}(b) = a, \exists s' \in S, \delta(s, b, s')\}$ 
         $i = 1$ 
        for each  $b \in Q$ 
             $\lambda = \lambda \cup \{(s, a \wedge nd = i, \mathbf{mkd}(b))\}$ 
            if  $\delta(s, b, s') = \text{true}$  then
                 $R = R \cup \{(s, a \wedge nd = i, s')\}$ 
             $i = i + 1$ 
        choose some  $b \in Q$ 
        for  $j=i$  to  $\text{MAX}$ 
             $\lambda = \lambda \cup \{(s, a \wedge nd = j, \mathbf{mkd}(b))\}$ 
             $R = R \cup \{(s, a \wedge nd = j, s')\}$ 
    
```

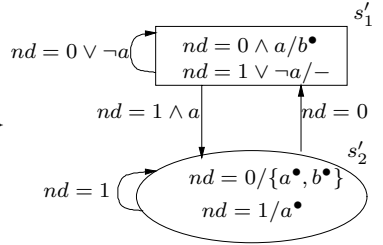


Fig. 9. A deterministic Mealy machine

Fig. 8. Pseudo-code for **GenMealy**

Example 8. The Mealy machine obtained from the automaton in Figure 9 is shown in Figure 10. nd is the new unconstrained internal signal. The labelling in state s_2 denotes the input/output function, for example $nd = 1/a^\bullet$ represents input $nd = 1$ and output is $a^\bullet = 1$.

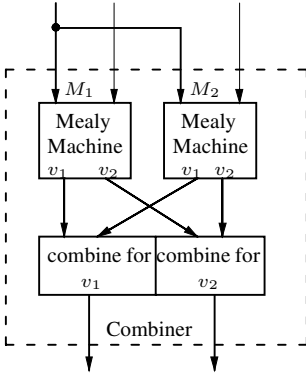


Fig. 10. Combiners for a Module

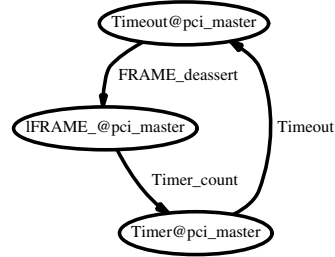


Fig. 11. Identified Combinational dependency loops

Theorem 2. *Let S be an ESL script. Then the language of the synthesized Mealy machine coincides with $\text{Traces}(S)$ on global variables. Formally,*

$$\text{clean}(\text{Traces}(\text{ModuleSynthesis}(S))) \approx_{\text{gl}} \text{Traces}(S).$$

The obtained Mealy machine can be translated into a Verilog program easily. Details are omitted due to the space restrictions.

4.3 Combiner Generation

Recall that each ESL script S is translated into a Mealy machine, and that different scripts can assign values to the same signal. In Section 4 we defined the operation \bowtie to combine two traces. As exemplified in Example 4, the operation \bowtie on traces is not the same as conjunction of scripts.

On the level of Mealy machines, we introduce *combiners* which perform the operation \bowtie on traces. A combiner machine is defined as follows.

Definition 8. Given a sequence M_1, \dots, M_m of Mealy machines with possibly nondisjoint alphabets, the *combiner* $C(M_1, \dots, M_m)$ is a Mealy machine whose input alphabet is the union of the input alphabets of the M_i and whose output alphabet is obtained from the union of the output alphabets of the M_i by removing the marks. On input of a symbol σ , $C(M_1, \dots, M_m)$ simulates each of the Mealy machines, collects the outputs o_1, \dots, o_m of the Mealy machines, and nondeterministically outputs one element of $\text{clean}(\text{comp}(o_1 \bowtie o_2 \bowtie \dots \bowtie o_m))$.

Thus, if the output of the Mealy machines is consistent, the combiner will output it; if it is inconsistent, then the combiner will output $\{false\}$; if the output does not determine the values of all signals, the combiner will nondeterministically choose consistent values for the unspecified or under-constrained signals.

The algorithm **Combine** generates Verilog code for the combiner. In the Verilog translation, each Mealy machine is translated into a Verilog module. The combiner itself

is a module which uses the Mealy machine modules as subprograms and combines their outputs.

In the traditional approach (i.e., synthesizing the conjunction of *all* scripts), the number of tableau states for the conjunction of the scripts can become exponentially larger. In fact, our techniques are tailored to find inconsistencies between scripts; in conjoined formulas, inconsistent behaviors would be eliminated.

Finally, we can formally state the correctness of our algorithms, cf. Section [4](#).

Theorem 3. *Let $\mathcal{E} = S_1 \parallel \dots \parallel S_n$ be an ESL module. Then*

$$\mathcal{L}(\text{ModuleSynthesis}(\mathcal{E})) \approx_{\text{gl}} \text{Traces}(S_1) \bowtie \dots \bowtie \text{Traces}(S_n).$$

4.4 Module Connection

In Verilog, connecting the inputs and outputs of modules can be done hierarchically. For synchronous bus designs, combinational loops are not allowed. Therefore, in the variable dependency graph, we use standard algorithms for strongly-connected components [\[2\]](#) to identify combinational loops. Our tool will report combinational loops to the users. Figure [1](#) shows an example combinational loop identified during PCI bus protocol debugging. In the Figure, *Timeout*, *lFRAME_*, and *Timer* are signals in module *pci_master*, and *FRAME_deassert*, *Timer_count* and *Timeout* are names of scripts in this module. Each edge represents a dependency between the two signals introduced by the script, e.g., signal *lFRAME_* depends on *Timeout* in script *FRAME_deassert*.

5 Debugging Protocols in ESL

For a protocol specified in ESL, it is essential to have debugging capabilities which verify that the specified protocol is indeed what the designers want to specify. In this section, we describe special properties of the generated Verilog and SMV code which facilitate debugging. Due to space restrictions, we describe only some of the debugging capabilities which we implemented. The code generator for Verilog and SMV can be extended easily to handle other capabilities.

Synchronization Contradiction. In ESL, two scripts can potentially assert contradictory values to signals, cf. Example [1](#). To detect such cases, a flag *OC* is defined in the Verilog code for the combiner. As soon as the combiner computes $\{\text{false}\}$, the flag is automatically set to 1. According to Theorem [1](#), the flag *OC* is set to 1 if and only if the traces of the scripts contradict. Therefore, to verify the absence of synchronization contradictions (i.e., consistency) it suffices to check for unreachability of $OC = 1$.

Receptiveness. A machine is receptive [\[2\]](#) in an environment, if starting from any state, for all possible inputs from the environment, the machine can produce valid output and transit to a valid next state. Receptiveness is among the most important properties of protocols. In ESL, receptiveness questions arise on the level of *modules*. Receptiveness is interesting for a module in connection with the whole protocol, but also for a single module with unconstrained inputs. For each module *M*, a special flag BI_M is defined

in the Verilog code for M . If a Mealy machine belonging to M gets stuck, i.e., has no valid transition for the given input, then the flag BI_M is set to 1.

The absence of synchronization contradictions and the unreachability of BI_M for each M guarantees the receptiveness of the ESL protocol. Deadend checks for receptiveness have been proposed previously for Moore machines in [12].

Property Checking. The specification of protocols often can be partitioned into specifications of a fundamental and operational character, and another set of additional specifications which were logically redundant for a correct protocol, but are aimed at finding out if the protocol is correctly specified.

In ESL, we build Verilog and SMV models based on the operational specifications, and then use a Verilog simulator and the SMV model checker to verify the additional specifications.

6 Experimental Results

We have built a prototype system in Standard ML, which includes about 13,000 lines of code. To test the capability of our tool, we have specified a subset of PCI bus protocol [13]. The specification consists of 118 formulas and 1028 lines including English comments for each formula. The specification includes five module types: master sequencer, master backend, target sequencer, target backend, and arbiter. One master sequencer and one master backend form a PCI master; conversely, one target sequencer and one target backend form a PCI target. We have specified a biased random test bench [14] in the master backend, because in PCI, the test bench involves transaction request generation, which is an integral part of the behavior of the master backend. Using the parameter concept in ESL, the number of modules on the bus can be easily modified. Independently, [15] developed a specification methodology for the PCI protocol, in which explicitly declared signals are used for monitoring protocol violation.

The specified subset of the PCI protocol includes burst transaction, master-initiated fast-back-to-back transaction, master and target termination conditions, initial and subsequent data latencies, target decode latency, master reissue of retried request. We are currently working on configuration cycles, bus parking and parity checking. We plan to specify the complete PCI bus protocol in future work.

Our algorithm is very efficient, and it only takes about 15 seconds to generate the SMV or Verilog model from 62 ESL scripts on a 550MHz Pentium III machine with 256 MB memory. The generated Verilog is about 7000 lines of code while the generated SMV is about 6100 lines of code. Using the Cadence Verilog-XL simulator, we are able to simulate the generated Verilog model. Many easy errors have been identified by checking synchronization contradiction in Verilog simulation, including some errors in the protocol. For example, the following two statements from [15] can be contradictory when a PCI target can perform fast decode.

1. In page 26, A PCI target is required to assert its TRDY# signal in a read transaction unconditionally when the data is valid.
2. In page 47, The first data phase on a read transaction requires a turnaround-cycle (enforced by the target via TRDY#).

The sixty nine Verilog monitors from [12] are used to characterize correctness of the PCI bus protocol. In order to verify that our Verilog model is correct, we connect the monitors with our model to check whether our model violates the monitors.

Model checking can formally verify the correctness of a given formula. However, it is limited by the size of the model. We use it on a limited configuration of the protocol, namely one master, one target, one dummy arbiter. After abstracting the data path, e.g., the width of the bus, we have successfully model checked 70 temporal properties which are derived from the temporal properties given by [12]. The verification takes 450MB memory and 2 minutes on a 360MHz Sun 6500 Enterprise server.

7 Conclusions

We have proposed a new logic ESL for formal specification of hardware protocols. Our synthesis methodology generates executable code in Verilog or SMV from ESL specifications. A significant part of the PCI bus protocol has been specified, simulated, and verified using our tool. Our experimental results clearly demonstrate that our synthesis methodology is feasible for systems of realistic complexity.

In the future, we plan to complete the PCI bus specification and to experiment with other industrial bus protocols such as the IBM CoreConnect bus and the Intel P6 bus. To achieve this goal, we are investigating various extensions of ESL. In particular, we believe it is possible to incorporate our logic into Verilog without significantly changing the syntax of the language. Such an extension should also make our tool easier for engineers to use.

Finally, we believe that game theoretic notions of consistency [13, 14] are necessary under certain circumstances. We intend to investigate various notions of consistency for our logic and devise algorithms for verifying them.

Acknowledgment

We are grateful to David Dill and Kanna Shimizu for discussions and comments on our paper, and for providing the PCI properties [12].

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar and Y. Wolfsthal, "FoCs - Automatic Generation of Simulation Checkers from Formal Specifications", *CAV 00: Computer-Aided Verification*, Lecture Notes in Computer Science 1855, 538-542. Springer-Verlag, 2000.
2. R. Alur and T.A. Henzinger. Reactive Modules, *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, 207-218. IEEE Computer Society Press, 1996.
3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, 100-109, 1997.
4. R. E. Bryant, P. Chauhan, E. M. Clarke, A. Goel. A Theory of Consistency for Modular Synchronous Systems. *Formal Methods in Computer-Aided Design*, 2000
5. P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-Core based System-On-Chip Designs. In *Proceedings of the IEEE ASIC Conference*, 27-31, 1999.

6. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithm. *MIT Press*, 1990.
7. E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs: Workshop*, Yorktown Heights, NY, May 1981 Lecture Notes in Computer Science, Vol. 131, Springer-Verlag. 1981.
8. E. Emerson and E. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. In *Science of Computer Programming*, Vol 2, 241-266. 1982.
9. E. Clarke, O. Grumberg, and D. Peled. Model Checking. *MIT Publishers*, 1999.
10. David Dill. Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits. *MIT Press*, 1989.
11. M. Fujita and S. Kono. Synthesis of Controllers from Interval Temporal Logic Specification. *International Workshop on Logic Synthesis*, May, 1993.
12. D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, Vol. 398, 409-448. Springer Verlag, LNCS 398, 1989.
13. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
14. N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Sixth International Symp. on Lucid and Intensional Programming*, ISLIP'93, Quebec City, Canada, April 1993. Universit'e Laval.
15. L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872-923, March 1994.
16. Z. Manna, P. Wolper: Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM TOPLAS*, Vol.6, N.1, Jan. 1984, 68-93.
17. K.L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.
18. B. Moszkowski. Executing temporal logic programs. *Cambridge University Press*, 1986.
19. PCI Special Interest Group. PCI Local Bus Specification Rev 2.2. Dec. 1998.
20. J. Yuan, K. Shultz, C. Pixley, and H. Miller. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *International Conference on Computer-Aided Design*. 584-589, November 7-11, 1999
21. A. Seawright, and F. Brewer. Synthesis from Production-Based Specifications. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 194-199, 1992.
22. X. Shen, and Arvind. Design and Verification of Speculative Processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, June 1998, Marstrand, Sweden.
23. K. Shimizu, D. Dill, and A. Hu. Monitor Based Formal Specification of PCI. *Formal Methods in Computer-Aided Design*, 2000.
24. K. Shimizu. <http://radisn.stanford.edu/pci>
25. M. Sipser. Introduction to the Theory of Computation. *PWS Publishing Company*. 1997.

Formal Verification of Floating Point Trigonometric Functions

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway
Hillsboro, OR 97124, USA

Abstract. We have formal verified a number of algorithms for evaluating transcendental functions in double-extended precision floating point arithmetic in the Intel® IA-64 architecture. These algorithms are used in the Itanium™ processor to provide compatibility with IA-32 (x86) hardware transcendentals, and similar ones are used in mathematical software libraries. In this paper we describe in some depth the formal verification of the sin and cos functions, including the initial range reduction step. This illustrates the different facets of verification in this field, covering both pure mathematics and the detailed analysis of floating point rounding.

1 Introduction

Code for evaluating the common transcendental functions (sin, exp, log etc.) in floating point arithmetic is important in many fields of scientific computing. For simplicity and portability, various standard algorithms coded in C are commonly used, e.g. those distributed as part of FDLIBM (Freely Distributable LIBM). However, to achieve better speed and accuracy, Intel is developing its own mathematical algorithms for the new IA-64 architecture, hand-coded in assembly language.

Many of these algorithms are complex and their error analysis is intricate. Subtler algorithmic errors are difficult to discover by testing. In order to provide improved quality assurance, Intel is subjecting the most important algorithms to formal verification. In particular, the IA-64 architecture provides full compatibility with IA-32 (x86), and the latter includes a small set of special instructions to evaluate core transcendental functions. These will be the focus here, though essentially the same techniques can be used in other contexts.

Many of the algorithms for evaluating the transcendental functions follow a similar pattern. However, IA-64 supports several floating-point precisions: single precision (24 significand bits), double precision (53) and double-extended precision (64), and the target precision significantly affects the design choices. Since internal computations can be performed in double-extended precision, rounding errors are much less a concern when the overall computation is for single or double precision. It is relatively easy to design simple, fast and accurate algorithms

¹ See <http://www.netlib.org/rdlib/>.

of the sort Intel provides [10]. For double-extended precision functions — such as the IA-32 hardware transcendentals — much more care and subtlety is required in the design [11] and the formal verifications are significantly more difficult.

In the present paper, to avoid repetition and dilution, we focus on the formal verification of an algorithm for a particular pair of functions: the double-extended floating point sine and cosine. This is used for compatibility with the IA-32 hardware transcendentals **FSIN**, **FCOS** and **FSINCOS**. Essentially the same algorithm, with the option of a more powerful initial range-reduction step for huge input arguments, is used in Intel’s double-extended precision mathematical library that can be called from C and FORTRAN. These particular functions were chosen because they illustrate well the many aspects of the formal verifications, involving as they do a sophisticated range reduction step followed by a tricky computation carefully designed to minimize rounding error. They are somewhat atypical in that they do not use a table lookup [12], but otherwise seem to show off most of the interesting features.

2 Outline of the Algorithm

The algorithm is intended to provide accurate double-extended approximations for $\sin(x)$ and $\cos(x)$ where x is a double-extended floating point number in the range $-2^{63} \leq x \leq 2^{63}$. (Although there are separate entry points for sine and cosine, most of the code is shared and both sine and cosine can be delivered in parallel, as indeed is required by **FSINCOS**.) According to the IA-32 documentation, the hardware functions just return the input argument and set a flag when the input is out of this range. The versions in the mathematical library, however, will reduce even larger arguments. An assembler code implementation of the mathematical library version is available on the Web, and this actual code can be examined as a complement to the more abstract algorithmic description given here.

<http://developer.intel.com/software/opensource/numerics/index.htm>

The algorithm is separated into two phases: an initial range reduction, and the core function evaluation. Mathematically speaking, for any real number x we can always write:

$$x = N(\pi/2) + r$$

where N is an integer (the closest to $x \cdot \frac{2}{\pi}$) and $|r| \leq \pi/4$. We can then evaluate $\sin(x)$ and/or $\cos(x)$ as either $\sin(r)$, $\cos(r)$, $-\sin(r)$ or $-\cos(r)$ depending on N modulo 4. For example:

$$\sin((4M + 3)(\pi/2) + r) = -\cos(r)$$

We refer to the process of finding the appropriate r and N (modulo 4 at least) as *trigonometric range reduction*. The second phase, the core evaluation, need now only be concerned with evaluating $\sin(r)$ or $\cos(r)$ for r in a limited

range, and then perhaps negating the result, depending on N modulo 4. Moreover, since $\cos(x) = \sin(x + \pi/2)$ we can perform the same range reduction and core evaluation for $\sin(x)$ and $\cos(x)$, merely adding 1 to N at an intermediate stage if $\cos(x)$ is required. Thus, as hinted earlier, the sine and cosine functions share code almost completely, merely setting an initial value of N to be 0 or 1 respectively.

The core evaluation of $\sin(r)$ and $\cos(r)$ can now be performed using a power series expansion similar to a truncation of the familiar Taylor series:

$$\begin{aligned}\sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \\ \cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots\end{aligned}$$

but with the pre-stored coefficients computed numerically to minimize the maximum error over r 's range, using the so-called *Remez algorithm* [10]. The actual evaluations of the truncated power series in floating point arithmetic, however, require some care if unacceptably high rounding errors are to be avoided.

3 HOL Floating Point Theory

The verification described here is conducted in the HOL Light theorem prover [11], and the formal proofs are founded on formalized HOL theories of mathematical analysis [12] and floating point arithmetic [13]. Because of space limitations, we cannot describe these theories in great detail here, but we will sketch a few highlights, particularly of the floating point material where there is less established notation. We hope this will suffice for the reader to follow the explicit HOL theorems given below.

HOL Light is a highly foundational theorem proving system using the methodology first established by Edinburgh LCF [14]. ‘LCF style’ provers explicitly generate proofs in terms of extremely low-level primitive inferences, in order to provide a high level of assurance that the proofs are valid. In HOL Light, as in most other LCF-style provers, the proofs (which can be very large) are not usually stored permanently, but the strict reduction to primitive inferences is maintained by the abstract type system of the interaction and implementation language, which for HOL Light is CAML Light [15, 16]. The primitive inference rules of HOL Light, which implements a simply typed classical higher order logic, are very simple. However CAML Light also serves as a programming medium allowing higher-level derived rules (e.g. to automate linear arithmetic, first order logic or reasoning in other special domains) to be programmed as automatic reductions to primitive inferences. This lets the user conduct the proof at a more palatable level, while still maintaining the logical safety that comes from low-level proof generation. A few application-specific instances of programming derived rules in HOL Light will be described in the present paper.

HOL's foundational style is also reflected in its approach to developing new mathematical theories. All HOL mathematics is developed by *constructing* new

structures from the primitive logical and set theoretic basis, rather than by asserting additional axioms. For example, the natural numbers are constructed by inductively carving out an appropriate subset of the infinite set asserted to exist by the basic Axiom of Infinity. In turn, such inductive definitions are defined as appropriate set-theoretic intersections, rather than being permitted as primitive extensions. The positive real numbers are defined as equivalence classes of nearly-additive functions $\mathbb{N} \rightarrow \mathbb{N}$ (equivalent to a version of Cantor's construction using Cauchy sequences, but without explicit use of rationals), and reals as equivalence classes of pairs of positive real numbers. After the development of some classical analytical theories of limits, differentiation, integration, infinite series etc., the most basic transcendental functions (exp, sin and cos) are defined by their power series and proved to have their standard properties. Some 'inverse' transcendental functions like \ln and atan are defined abstractly and their properties proven via the inverse function theorem. For more details, the reader can consult [10].

HOL notation is generally close to traditional logical and mathematical notation. However, the type system distinguishes natural numbers and real numbers, and maps between them by `&`; hence `&2` is the real number 2. The multiplicative inverse x^{-1} is written `inv(x)` and the power x^n as `x pow n`. Note that the expression `Sum(m,n) f` denotes $\sum_{i=m}^{m+n-1} f(i)$, and not as one might guess $\sum_{i=m}^n f(i)$.

Much of the theory of floating point numbers is generic. Floating point formats are identified by triples of natural numbers `fmt` and the corresponding set of representable real numbers, ignoring the upper limit on the exponent range, is `iformat fmt`. The second field of the triple, extracted by the function `precision`, is the precision, i.e. the number of significant bits. The third field, extracted by the `ulp scale` function, is N where 2^{-N} is the smallest nonzero floating point number of the format.

Floating-point rounding is performed by `round fmt rc x` which denotes the result of rounding the real number `x` into `iformat fmt` (the representable numbers, with unlimited exponent range, of a floating point format `fmt`) under rounding mode `rc`. The predicate `normalizes` determines whether a real number is within the range of normal floating point numbers in a particular format, i.e. those representable with a leading 1 in the significand, while `losing` determines whether a real number will lose precision, i.e. underflow, when rounded to a given format.

Most of the difficulty of analyzing floating point algorithms arises from the error committed in rounding floating point results to fit in their destination floating point format. The theorems we use to analyze these errors can be subdivided into (i) routine worst-case error bound theorems, and (ii) special cases where no rounding error is committed.

3.1 The $(1 + \epsilon)$ Property

In typical error analyses of high-level floating point code, the first kind of theorem is used almost exclusively. The mainstay of traditional error analysis, often called the ' $(1 + \epsilon)$ ' property, is simply that the result of a floating point operation is the

exact result, perturbed by a relative error of bounded magnitude. Recalling that in our IEEE arithmetic, the result of an operation is the rounded exact value, this amounts to saying that x rounded is always of the form $x(1 + \epsilon)$ with $|\epsilon|$ bounded by a known value, typically 2^{-p} where p is the precision of the floating point format. We can derive a result of this form fairly easily, though we need sideconditions to exclude the possibility of underflow (not overflow, which we consider separately from rounding). The main theorem is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
  ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
    (round fmt rc x = x * (&1 + e))
```

This essentially states exactly the ‘ $1+\epsilon$ ’ property, and the bound on ϵ depends on the rounding mode, according to the following auxiliary definition of `mu`:

```
|- (mu Nearest = &1 / &2) ∧ (mu Down = &1) ∧
  (mu Up = &1) ∧ (mu Zero = &1)
```

The theorem has two sideconditions, the second being that the floating point format is not trivial (it has a nonzero number of fraction bits), and the first being an assertion that the value x does not *lose precision*, in other words, that the result of rounding x would not change if the lower exponent range were extended. We will not show the formal definition [\[11\]](#) here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range or is zero:

```
|- normalizes fmt x ⇒ ¬(losing fmt rc x)
```

where

```
|- normalizes fmt x =
  (x = &0) ∨
  &2 pow (precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(x)
```

There is also a similar theorem for absolute rather than relative error analysis, which is sometimes useful for obtaining sharper error bounds, but will not be shown here. It does not require any normalization hypotheses, which can make it simpler to apply.

3.2 Cancellation Theorems

In lower-level algorithms like the ones considered here and others that the present author is concerned with verifying, a number of additional properties of floating point arithmetic are sometimes exploited by the algorithm designer and proofs of them are required for verifications. In particular, there are important situations where floating point arithmetic is exact, i.e. results round to themselves. This happens if and only if the result is representable as a floating point number:

```
|- a ∈ iformat fmt ⇒ (round fmt rc a = a)
```

```
|- ¬(precision fmt = 0) ⇒ ((round fmt rc x = x) = x ∈ iformat fmt)
```

There are a number of situations where arithmetic operations are exact. Perhaps the best-known instance is subtraction of nearby quantities; cf. Theorem 4.3.1 of [14]:

```
|- a ∈ iformat fmt ∧ b ∈ iformat fmt ∧ a / &2 ≤ b ∧ b ≤ &2 * a
   ⇒ (b - a) ∈ iformat fmt
```

Another classic result [14] shows that we can obtain the sum of two floating point numbers exactly in two parts, one a rounding error in the other, by performing the floating point addition then subtracting both summands from the result, the larger one first:

```
|- x ∈ iformat fmt ∧
   y ∈ iformat fmt ∧
   abs(x) ≤ abs(y)
   ⇒ (round fmt Nearest (x + y) - y) ∈ iformat fmt ∧
      (round fmt Nearest (x + y) - (x + y)) ∈ iformat fmt
```

As we will see later, theorems of this sort, including some rather *ad hoc* derivative lemmas, are used extensively in the analysis of trigonometric range reduction, where almost every floating point operation is based on some special trick to avoid rounding error or later compensate for it! Some of these results are not readily found in the literature, but are well-known to experts in floating point arithmetic. Sometimes the lemmas we have ended up proving are not optimal and could profitably be sharpened, but having performed quite a few verifications we are confident that we have a fairly comprehensive basic toolkit.

4 Verification of Range Reduction

The principal difficulty of implementing trigonometric range reduction is that the input argument x may be large and yet the reduced argument r very small, because x is unusually close to a multiple of $\pi/2$. In such cases, the computation of r needs to be performed very carefully. Assuming we have calculated N , we need to evaluate:

$$r = x - N \frac{\pi}{2}$$

However, $\frac{\pi}{2}$ is irrational and so cannot be represented exactly by any finite sum of floating point numbers. So however the above is computed, it must in fact calculate

$$r' = x - NP$$

for some approximation $P = \frac{\pi}{2} + \epsilon$. The relative error $\frac{|r' - r|}{|r|}$ is then $\frac{N|\epsilon|}{|r|}$ which is of the order $|\frac{x\epsilon}{r}|$. Therefore, to keep this relative error within acceptable bounds (say 2^{-70}) the accuracy required in the approximation P depends on how small the (true) reduced argument can be relative to the input argument. In order to formally verify the accuracy of the algorithm, we need to answer the purely mathematical question: how close can a double-extended precision floating point number be to an integer multiple of $\frac{\pi}{2}$? Having done that, we can proceed with the verification of the actual computation of the reduced argument in floating point arithmetic.

4.1 Approximating π

For the proofs that follow we need to have an accurate rational approximation to π , and of course a *formal proof* that it is sufficiently accurate. The accuracy we need (2^{-225}) follows from the later proofs, but we first wrote a routine to approximate π to arbitrary precision, since less accurate approximations are useful for disposing of trivial sideconditions that crop up in proofs, e.g. $1 < \pi/2$.

Our starting point for approximating π is the Taylor series for the arctangent, which we had already derived for use in the verification of floating point arctangent functions. The proof of this proceeds as follows, using some of the real analysis described in [1]. We demonstrate, using the comparison test and the pre-proved convergence of the geometric series $\sum_{m=0}^{\infty} x^m$ for $|x| < 1$, that the infinite series $\sum_{m=0}^{\infty} \frac{(-1)^m}{2m+1} x^{2m+1}$ converges for $|x| < 1$ to some limit function $f(x)$. Therefore the series can provably be differentiated term-by-term, and the derivative series is $\sum_{m=0}^{\infty} (-1)^m (x^2)^m$ which is again a geometric series and sums to $\frac{1}{1+x^2}$. Consequently $f'(x)$ and $\tan'(x) = \frac{1}{1+x^2}$ coincide for $|x| < 1$, and so $\text{atn}(x) - f(x)$ is constant there. But for $x = 0$ we have $\text{atn}(x) - f(x) = 0$, and hence it follows that the series converges to $\text{atn}(x)$ for all $|x| < 1$. The error in truncating the series can trivially be bounded provided $|x| \leq 1/2^k$ for $k > 0$, giving us the final theorem which in HOL looks like this:

```
|- abs(x) <= inv(&2 pow k) ^ (k = 0)
  => abs(atn x -
    Sum(0,n) (\m. (if EVEN m then &0
                  else --(&1) pow ((m - 1) DIV 2) / &m) *
                  x pow m))
    <= inv(&2 pow (n * k - 1))
```

It is now easy to obtain approximations to π by applying atn to appropriate rational numbers. We wrote a HOL derived rule `MACHIN_RULE` that computes (with proofs) linear forms in arctangents of rational numbers using the addition formula:

```
|- abs(x * y) < &1 => (atn(x) + atn(y) = atn((x + y) / (&1 - x * y)))
```

The user can apply `MACHIN_RULE` to any linear form in arctangents, and it is automatically decomposed into sums and negations, and the above theorem

used repeatedly to simplify it, with the side-condition $|xy| < 1$ discharged automatically at each stage. In this way, useful approximation theorems that are variants on the classic Machin formula can be derived automatically without the user providing separate proofs:

```
#let MACHIN_1 = MACHIN_RULE '&4 * atn(&1 / &5) - atn(&1 / &239)';;
MACHIN_1 : thm = |- pi / &4 = &4 * atn (&1 / &5) - atn (&1 / &239)
#let STRASSNITZKY_MACHIN = MACHIN_RULE
  'atn(&1 / &2) + atn (&1 / &5) + atn(&1 / &8)';;
STRASSNITZKY_MACHIN : thm =
  |- pi / &4 = atn (&1 / &2) + atn (&1 / &5) + atn (&1 / &8)
#let MACHINLIKE_1 = MACHIN_RULE
  '&6 * atn(&1 / &8) + &2 * atn(&1 / &57) + atn(&1 / &239)';;
MACHINLIKE_1 : thm =
  |- pi / &4 = &6 * atn (&1 / &8) + &2 * atn (&1 / &57) + atn (&1 / &239)
```

We use the last of these to derive approximations to π , again via a derived rule that for any given accuracy returns an approximation of π good to that accuracy. For example:

```
#let pth = PI_APPROX_RULE 5;;
pth : thm = |- abs (pi - &26696452523 / &8498136384) <= inv (&2 pow 5)
```

The above approach is easy to prove and program up, and adequate for the accuracies we needed for this proof, but for more precise approximations to π , we would probably need to exploit more efficient approximation methods for π such as the remarkable series [\[1\]](#) which we have already formally verified in HOL:

$$\pi = \sum_{m=0}^{\infty} \frac{1}{16^m} \left(\frac{4}{8m+1} - \frac{2}{8m+4} - \frac{1}{8m+5} - \frac{1}{8m+6} \right)$$

4.2 Bounding the Reduced Argument

Armed with the ability to find arbitrarily good rational approximations to π , we can now tackle the problem: how close can our input number be to a nonzero integer multiple of $\pi/2$? Every double-extended precision floating point number x with $|x| < 2^{64}$ can be written

$$x = k/2^e$$

for some integers $2^{63} \leq k < 2^{64}$ and $e > 0$; assuming the number x is normalized, k is simply its significand considered as an integer. We are then interested in bounding:

$$\begin{aligned} & |k/2^e - N\pi/2| \\ &= \frac{|N|}{2^e} (k/N - 2^e\pi/2) \end{aligned}$$

We only need to consider cases where $k/2^e \approx N\pi/2$ since otherwise x is not close to a multiple of $\pi/2$. So we can consider only N such that $|N| \leq 2^{65-e}/3.14159$. Moreover, we need only consider $e < 64$ since otherwise $|x| < 1$ and so x is too small to be close to a nonzero multiple of $\pi/2$. So, for each $e = 0, 1, \dots, 63$ we just need to find the closest rational number p/q to $2^e\pi/2$ with $|q| \leq 2^{65-e}/3.14159$. We can then get a reasonable lower bound for $|k/2^e - N\pi/2|$ by:

$$\frac{2^{63-2e}}{3.1416} |p/q - 2^e\pi/2|$$

This is not quite optimal: we could use our knowledge of p and q to avoid the overestimate contained in the factor on the left, and rely on the fact that the term on the right would be significantly larger again for other p'/q' . However it is good to within a factor of 2, enough for our purposes.

We now have merely to solve (63 instances of) a classic problem of diophantine approximation: how close can a particular real number x be approximated by rational numbers p/q subject to some bound on the size of q ? There is a well-established method for solving this problem, which is easy to formalize in HOL. Suppose we have two straddling rational numbers $p_1/q_1 < x < p_2/q_2$ such that $p_2q_1 = p_1q_2 + 1$. It is easy to show that any other rational approximation a/b with $b < q_1 + q_2$ is no better than the closer of p_1/q_1 and p_2/q_2 . In such a case, unless p_1/q_1 and a/b are the same rational we must have $|p_1/q_1 - a/b| = |p_1b - q_1a|/(q_1b) \geq 1/(q_1b)$, since the numerator $p_1b - q_1a$ is a nonzero integer. Similarly, $|p_2/q_2 - a/b| \geq 1/(q_2b)$ unless $p_2/q_2 = a/b$. Therefore, since $|b| < q_1 + q_2$:

$$\begin{aligned} |p_1/q_1 - a/b| + |p_2/q_2 - a/b| &\geq 1/(q_1b) + 1/(q_2b) \\ &> 1/(q_1q_2) \\ &= |p_1/q_1 - p_2/q_2| \end{aligned}$$

Consequently a/b cannot lie inside the straddling interval, and so cannot be closer to x . This is easily proved in HOL:

```
|- (p2 * q1 = p1 * q2 + 1) ^ ¬(q1 = 0) ^ ¬(q2 = 0) ^
  (&p1 / &q1 < x ^ x < &p2 / &q2)
  ==> ∀ a b. ¬(b = 0) ^ b < q1 + q2
    ==> abs(&a / &b - x) >= abs(&p1 / &q1 - x) ∨
        abs(&a / &b - x) >= abs(&p2 / &q2 - x)
```

It remains to find these special straddling pairs of rational numbers, for it is not immediately obvious that they must exist. Note that the finding process does not need to produce any kind of proof; the numbers can be found via an arbitrary method and the property checked formally by plugging the numbers into the above theorem. The most popular method for finding such ‘convergents’ uses continued fractions [2]. We use instead a procedure that is in general less

efficient but is simpler to program in our context, creating convergents iteratively by calculating the *mediant* of two fractions.

If we have two fractions p_1/q_1 and p_2/q_2 with $p_2q_1 = p_1q_2 + 1$ (and hence $p_1/q_1 < p_2/q_2$) then it is easy to prove that the mediant p/q where $p = p_1 + p_2$ and $q = q_1 + q_2$ has the same property with respect to its parents: $pq_1 = p_1q + 1$ and $p_2q = pq_2 + 1$. Note that this implies $p_1/q_1 < p/q < p_2/q_2$ and that p/q is already in its lowest terms (any common factor of p and q would, since $p_2q = pq_2 + 1$, divide 1). In fact, iterating this generative procedure starting with just $0/1$ and $1/1$ generates all rational numbers between 0 and 1 in their lowest terms; this can be presented as the *Farey sequence* or *Stern-Brocot tree* [1].

We can now easily generate convergents to any real number x by binary chop: if we have $p_1/q_1 < x < p_2/q_2$ with $p_2q_1 = p_1q_2 + 1$, we simply form the mediant fraction p/q and iterate either with p_1/q_1 and p/q or with p/q and p_2/q_2 depending which side of the mediant x lies. It's easy to resolve this inequality via a sufficiently good rational approximation to x . We proceed until $q_1 + q_2$ reaches the bound we are interested in, and then plug the values into the main theorem, obtaining a lower bound on the quality of rational approximations to x . Finally, we use the very good rational approximation to π to get a good rational lower bound for the terms $p_1/q_1 - x$ and $p_2/q_2 - x$ in the conclusion. Iterating in this way for the various choices of e , we find our overall bound for how close the input number can come to a multiple of $\pi/2$: about $113/2^{76}$:

```
|- integer(N) ^& neg(N = &0) ^&
  a ∈ iformat (rformat Register) ^& abs(a) < &2 pow 64
  => abs (a - N * pi / &2) >= &113 / &2 pow 76
```

4.3 Analyzing the Reduced Argument Computation

The above theorem shows that, unless $N = 0$ in which case reduction is trivial, the reduced argument has magnitude at least around 2^{-69} . Assuming the input has size $\leq 2^{63}$, this means that an error of ϵ in the approximation of $\pi/2$ can constitute approximately a $2^{132}\epsilon$ relative error in r . Consequently, to keep the relative error down to about 2^{-70} we need $|\epsilon| < 2^{-202}$. Since a floating-point number has only 64 bits of precision, it would seem that we would need to approximate $\pi/2$ by four floating-point numbers P_1, \dots, P_4 and face considerable complications in keeping down the rounding error in computing $x - N(P_1 + P_2 + P_3 + P_4)$. However, using an ingenious technique called *pre-reduction* [2], the difficulties can be reduced. Certain floating point numbers are exceptionally close to exact multiples of 2π ; in fact slight variants of the methods used in the previous section can easily find such numbers. In the present algorithms, a particular floating point number P_0 is used with

$$|P_0 - 4178442\pi| < 2^{-63.5}$$

(not, incidentally, so very far from our lower bound). By initially subtracting off a multiple of P_0 , incurring a small error compared with subtracting off the corresponding even multiple of π , we then only need to deal with numbers of size

$< 2^{24}$ in the main code. (The accurate subtraction of multiples of P_0 is similar in spirit to the main computation we discuss below, and we will not discuss it here for reasons of space.) Therefore, even in the worst case, we can store a sufficiently accurate $\pi/2$ as the sum of three floating-point numbers $P_1 + P_2 + P_3$, where the magnitude of P_{n+1} is less than half the least significant bit of P_n .

As noted earlier, the actual computations in the trigonometric range reduction rely heavily on special tricks to avoid or compensate for rounding error. The computation starts by calculating $y = P'x$, where $P' \approx \frac{2}{\pi}$, and then rounding it to the nearest integer N . Because P' is not exactly $\frac{2}{\pi}$, and the computation of y commits a rounding error, N may not be the integer closest to $x\frac{2}{\pi}$. However, it is always sufficiently close that the next computation, $s = x - NP_1$ (computed using a fused multiply-accumulate instruction, rather than by a separate multiplication and subtraction) is exact, i.e. no rounding error is committed. We will not show the rather messy general theorem that we use to justify this.

However, because P_1 is not exactly $\pi/2$, s is not yet an adequate reduced argument. Besides, we must deliver the reduced argument in two pieces $r + c$ with $|c| \ll |r|$, since in general merely coercing the reduced argument into a single floating point number would introduce unacceptable errors.

Different paths are now taken, depending on whether $|s| < 2^{-33}$. The path where this is not the case is simpler, so we will discuss that; the other uses the same methods but is significantly more complicated. Since $|s| \geq 2^{-33}$, it is sufficiently accurate to use $P_1 + P_2$, and hence all we need to do is subtract NP_2 from s . However, we need to make sure that $|c| \ll |r|$ for the reduced argument $r + c$ (the later computation would otherwise be inaccurate, since it neglects high powers of c in the power series), so we can't simply take $r = s$ and $c = -NP_2$. Instead we calculate:

$$\begin{aligned} w &= NP_2 \\ r &= s - w \\ c1 &= s - r \\ c &= c1 - w \end{aligned}$$

Apart from the signs (which are inconsequential, since rounding to nearest commutes with negation), the exactness of the all but the first line can be justified by the previous theorem on computing an exact sum, based on the easily proven fact that $|w| \leq 2^{-33} \leq |s|$. We have $r + c = s - w$ exactly, and so the only errors in the reduced argument are the rounding error in w and N times the error in approximating $\pi/2$ by $P_1 + P_2$, both of which are small enough for our purposes.

5 Verification of the Core Computation

The core computation is simply a polynomial in the reduced argument. If the reduced argument is sufficiently small, very short and quickly evaluated polynomials suffice. (Note that this tends to even out the overall runtime of the

algorithm, since it is exactly in the cases of small reduced arguments that the range reduction phase is more expensive.) In the extreme case where $|r| < 2^{-33}$, we can evaluate $\cos(r + c)$ by just $1 - 2^{-67}$, regardless of the value of r . (In round-to-nearest mode this always rounds to 1, so the computation looks pointless, but we need to give sensible results when rounding down or towards zero). In the most general case, when we know only that $|r|$ is bounded by about $\pi/4$, the polynomials needed are significantly longer. For example the most general *sin* polynomial used is of the form:

$$p(y) = y + P_1y^3 + P_2y^5 + \cdots + P_8y^{17}$$

where $y = r + c$ is the reduced argument, and the P_i are all floating point numbers. Note that the P_i are not the same as the coefficients of the familiar Taylor series (which in any case are not exactly representable as floating point numbers), but arrived at using the Remez algorithm to minimize the worst-case error over the possible reduced argument range. Evaluating all the terms with $y = r + c$ is unnecessarily complicated, since higher powers of c are negligible; instead the algorithm simply exploits the addition formula

$$\sin(r + c) = \sin(r) \cos(c) + \cos(r) \sin(c) \approx \sin(r) + c(1 - r^2/2)$$

and evaluates:

$$r + P_1r^3 + P_2r^5 + \cdots + P_8r^{17} + c(1 - r^2/2)$$

The overall error, apart from that in range reduction which in this case where the reduced argument is not small is almost negligible, is now composed of three components:

- The approximation error $|p(r + c) - \sin(r + c)|$.
- The additional error in neglecting higher powers of c : $|p(r + c) - (p(r) + c(1 - r^2/2))|$.
- The rounding error in actually computing $p(r) + c(1 - r^2/2)$.

It is straightforward to provably bound the second error using some basic real algebra and analysis. The approximation and rounding errors are more challenging.

5.1 Bounding the Approximation Error

We have implemented an automatic HOL derived rule to provably bound the error in approximating a mathematical function by a polynomial over a given interval. The user need only provide a derived rule to produce arbitrarily good Taylor series approximations over that interval. For example, for the *cos* function, we can easily derive the basic Taylor theorem:

```

|- abs(x) <= inv(&2 pow k)
  => abs(cos x -
    Sum(0,n) (\m. (if EVEN m
      then -- &1 pow (m DIV 2) / &(FACT m)
      else &0) * x pow m))
  <= inv(&(FACT n) * &2 pow (n * k))

```

It's then straightforward to package this up as a derived rule, which we call `MCLAURIN_COS_POLY_RULE`. Given natural numbers k and p , this will compute the required n , instantiate the theorem and produce, with a proof, a polynomial $p(x)$ such that:

$$\forall x. |x| \leq 2^{-k} \implies |\cos(x) - p(x)| \leq 2^{-p}$$

For example:

```

#MCLAURIN_COS_POLY_RULE 3 7;;
it : thm =
|- ∀x. abs x <= inv (&2 pow 3)
  => abs (cos x - poly [&1] x) <= inv (&2 pow 7)
#MCLAURIN_COS_POLY_RULE 2 35;;
it : thm =
|- ∀x. abs x <= inv (&2 pow 2)
  => abs(cos x - poly [&1; &0; --&1 / &2; &0; &1 / &24; &0;
    --&1 / &720; &0; &1 / &40320] x)
  <= inv(&2 pow 35)

```

For efficiency of exploration, shadow functions are also provided, which produce the polynomials as lists of numbers without performing proof, but in principle these are dispensable. In order to provably bound the accuracy of a polynomial approximation to any mathematical function, the only work required of the user is to provide these functions. For most of the common transcendental functions this is, as here, straightforward. Exceptions are the tangent and cotangent, for which arriving at the power series is a considerable amount of work.

In order to arrive at a bound on the gap between the mathematical function $f(x)$ and the polynomial approximation $p(x)$, the HOL bounding rule uses the Taylor series function to approximate $f(x)$ by a truncated Taylor series $t(x)$. If the bound is required to accuracy ϵ , the Taylor series is constructed so $|f(x) - t(x)| \leq \epsilon/2$ over the interval. Then, the remaining problem is to bound $|t(x) - p(x)|$ to the same accuracy $\epsilon/2$. Since $t(x) - p(x)$ is just a polynomial with rational coefficients, this part can be automated in a regular way. In fact, the polynomial-bounding routine can be used separately, and is used at another point in this proof. The approach used is a little different from that described in [25], though the way it is used in the proof is the same.

The fundamental fact underlying the polynomial bounding rule is that the maximum of a polynomial (as for any differentiable function) lies either at one of the endpoints of the interval or at a point of zero derivative. This is proved in

the following HOL theorem, which states that if a function f differentiable for $a \leq x \leq b$ has the property that $f(x) \leq K$ at all points of zero derivative, as well as at $x = a$ and $x = b$, then $f(x) \leq K$ everywhere.

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diff1 (f' x)) x) ∧
    f(a) <= K ∧ f(b) <= K ∧
    (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0) ⇒ f(x) <= K)
    ⇒ (∀x. a <= x ∧ x <= b ⇒ f(x) <= K)
```

This reduces our problem to finding the points of zero derivative, where the derivative is another polynomial with rational coefficients. (Of course, in practice we can only isolate the roots of the polynomial to arbitrary accuracy, rather than represent them with rational numbers. However as we shall see later it is easy to accommodate the imperfect knowledge.) This is done via a recursive scheme, based on another fundamental fact: for any differentiable function f , $f(x)$ can be zero only at one point between zeros of the derivative $f'(x)$. More precisely, if $f'(x) \neq 0$ for $a < x < b$ then if $f(a)f(b) \geq 0$ there are no points of $a < x < b$ with $f(x) = 0$:

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diff1 f'(x))(x)) ∧
    (∀x. a < x ∧ x < b ⇒ ¬(f'(x) = &0)) ∧
    f(a) * f(b) >= &0
    ⇒ ∀x. a < x ∧ x < b ⇒ ¬(f(x) = &0)
```

and on the other hand if $f(c)f(d) \leq 0$ for any $a \leq c \leq d \leq b$ then any point $a < x < b$ with $f(x) = 0$ must in fact have $c \leq x \leq d$:

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diff1 f'(x))(x)) ∧
    (∀x. a < x ∧ x < b ⇒ ¬(f'(x) = &0))
    ⇒ ∀c d. a <= c ∧ c <= d ∧ d <= b ∧
        f(c) * f(d) <= &0
        ⇒ ∀x. a < x ∧ x < b ∧ (f(x) = &0)
        ⇒ c <= x ∧ x <= d
```

Using this theorem, it is quite easy to isolate all points x_i with $f(x_i) = 0$ within arbitrarily small intervals with rational endpoints, and prove that this does indeed isolate all such points. We simply do so recursively for the derivative f' (since root isolation is trivial for constant or even linear polynomials, the recursion is well-founded). Conservatively, we can include all isolating intervals for f' 's zeros in f 's, to avoid the difficulty of analyzing inside these intervals; later these can be pruned. Now if two adjacent points with $f'(x_i) = 0$ and $f'(x_{i+1})$ have been isolated by $c_i \leq x_i \leq d_i$ and $c_{i+1} \leq x_{i+1} \leq d_{i+1}$, we need only consider — assuming $d_i < c_{i+1}$ since otherwise there is no question of a root between them — whether $f(d_i)f(c_{i+1}) \leq 0$. Since by the inductive hypothesis, there are no roots of f' inside the interval $d_i < x < c_{i+1}$, we can use the above theorems to find that there are either no roots or exactly one. If there is one, then (without proof) we can isolate it within a subinterval $[c, d]$ by binary chop

and then use the second theorem above to show that this isolates all roots with $d_i < x < c_{i+1}$.

Although this procedure is conservative, that doesn't matter for the use we will make of it — proving that all the roots have been isolated, even if we overcount, still makes the procedure of bounding the function over these isolating intervals a sound approach. Nevertheless, we do prune down the initial set when it is clear that the function could not cross zero within the interval. This follows naturally from interlocking recursions down the chain of derivatives $f, f', f'', \dots, f^{(n)}$ evaluating both bounds and isolating intervals for all zeros. The key theorem here is:

$$\begin{aligned} & |- (\forall x. a \leq x \wedge x \leq b \implies (f \text{ diff1 } (f' \ x)) \ x) \wedge \\ & (\forall x. a \leq x \wedge x \leq b \implies (f' \text{ diff1 } (f'' \ x)) \ x) \wedge \\ & (\forall x. a \leq x \wedge x \leq b \implies \text{abs}(f''(x)) \leq K) \wedge \\ & a \leq c \wedge c \leq x \wedge x \leq d \wedge d \leq b \wedge (f'(x) = 0) \\ & \implies \text{abs}(f(x)) \leq \text{abs}(f(d)) + (K / 2) * (d - c) \text{ pow } 2 \end{aligned}$$

which allows us to go from a bound on f'' and root isolation of f' to a bound on f (a sharp second-order one using the fact that the f' has a zero in the interval to show that f is flat there). As shown above, we can easily pass from root isolation of f' to root isolation of f . Hence we have a recursive procedure to bound f by recursively bounding and isolating f' and all additional derivatives. This is all done entirely automatically and HOL constructs a proof.

5.2 Bounding the Rounding Error

Most of the rounding errors in the polynomial computation can be bounded simply using the $(1+\epsilon)$ theorem or its absolute error analog. This is completely routine, and in fact has been automated [11]. However, there is one point in the sine and cosine calculations where a trick is used, and manual intervention in the proof is required. (However, for many other functions which don't use tricks, this part of the proofs *is* essentially automatic.) Without special measures, the rounding error in computing the second term of the series, $PP_1 r^3$ in the case of sine ($PP_1 \approx \frac{1}{3}$, or $QQ_1 r^2$ in the case of cosine ($QQ_1 = \frac{1}{2}$), would reduce the quality of the answer beyond acceptable levels. Therefore, these are computed in a more sophisticated way.

In order to reduce rounding error, r is split, using explicit bit-level operations, into two parts $r_{hi} + r_{lo}$, where r_{hi} has only 10 significant bits and r_{lo} is the corresponding “tail”. For cosine, the required $\frac{1}{2}r^2$ can be computed by:

$$\begin{aligned} \frac{1}{2}r^2 &= \frac{1}{2}r_{hi}^2 + \frac{1}{2}(r^2 - r_{hi}^2) \\ &= \frac{1}{2}r_{hi}^2 + \frac{1}{2}r_{lo}(r + r_{hi}) \end{aligned}$$

Because r_{hi} only has 10 significant digits, its squaring commits no rounding error, and of course neither does the multiplication by a power of 2. Thus, the

rounding error is confined to the much smaller quantity $r_{lo}(r + r_{hi})$, which is well within acceptable limits. For sine a similar trick is used, but since the coefficient is no longer a power of 2, it is also split. We will not show the details here.

6 Final Correctness Theorem

The final general correctness theorems we derive have the following form:

```
|- x ∈ floats Extended ∧ abs(Val x) ≤ &2 pow 64
  ⇒ prac (Extended,rc,fz) (fcos rc fz x) (cos(Val x))
    (#0.07341 * ulp(rformat Extended) (cos(Val x)))
```

The function **prac** means ‘pre-rounding accuracy’. The theorem states that provided x is a floating point number in the double-extended format, with $|x| \leq 2^{64}$ (a range somewhat wider than needed), the result excluding the final rounding is at most 0.07341 units in the last place from the true answer of $\cos(x)$. This theorem is generic over all rounding modes **rc** and flush-to-zero settings **fz**. An easy corollary of this is that in round-to-nearest mode without flush-to-zero set the maximum error is 0.57341 ulps, since rounding to nearest can contribute at most 0.5 ulps. In other rounding modes, a more careful analysis is required, paying careful attention to the formal definition of a ‘unit in the last place’. The problem is that the true answer and the computed answer before the final rounding may in general lie on opposite sides of a (negative, since $|\cos(x)| \leq 1$) power of 2. At this point, the gap between adjacent floating point numbers is different depending on whether one is considering the exact or computed result. In the case of round-to-nearest, however, this does not matter since the result will always round to the straddled power of 2, bringing it even closer to the exact answer.

7 Conclusions

We have presented a representative example of the work involved in verifications of this kind. As can be seen, the mathematical apparatus necessary for the verifications is quite extensive, and we require both abstract pure mathematics and very concrete results about floating point rounding. Moreover, since some parts of the proof, such as bounding approximation errors and routine rounding errors, would be extremely tedious by hand, programmability of the underlying theorem prover is vital. While these proofs are definitely non-trivial, modern theorem provers such as HOL Light have reached a stage of development (particularly in the formalization of the underlying mathematical theories) where they are quite feasible.

References

1. D. Bailey, P. Borwein, and S. Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66:903–913, 1997.
2. A. Baker. *A Consise Introduction to the Theory of Numbers*. Cambridge University Press, 1985.
3. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
4. T. J. Dekker. A floating-point technique for extending the available precision. *Numerical Mathematics*, 18:224–242, 1971.
5. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
6. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.
7. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
8. J. Harrison. Verifying the accuracy of polynomial approximations in HOL. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 137–152, Murray Hill, NJ, 1997. Springer-Verlag.
9. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
10. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.
11. J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4:1–7, 1999. This paper is available on the Web as http://developer.intel.com/technology/itj/q41999/articles/art_5.htm.
12. O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.
13. M. E. Remes. Sur le calcul effectif des polynômes d'approximation de Tchebichef. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 199:337–340, 1934.
14. P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.
15. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In I. Koren and P. Kornerup, editors, *Proceedings, 14th IEEE symposium on on computer arithmetic*, pages 4–11, Adelaide, Australia, 1999. IEEE Computer Society.
16. P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 232–236, 1991.
17. P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: <http://pauillac.inria.fr/caml/>.

Hardware Modeling Using Function Encapsulation

Jun Sawada and Warren A. Hunt, Jr.

IBM Austin Research Laboratory
sawada@us.ibm.com, whunt@us.ibm.com

Abstract. We describe how to specify an executable behavioral model of hardware without specifying the hardware detail using ACL2 encapsulation. ACL2 encapsulation is a mechanism to introduce abstract functions with constraints. It can be used to specify a microarchitectural design of hardware, which can be used for early simulation and for verification. Such a high-level design can also be used as a reference model when implementing low-level designs in RTL. This paper examines two abstract specifications from a microprocessor verification project. One example is a branch predictor for a processor with speculative execution and the other is a pipelined multiplier.

1 Introduction

We discuss high-level specifications of hardware using the function encapsulation mechanism of the ACL2 theorem prover [Sawada00]. ACL2 function encapsulation introduces abstract functions with certain constraints. We use this mechanism in order to define next-state functions which are both executable and verifiable, while we avoid specifying the detail of the machine design.

When specifying a high-level design of hardware, we want to be able to define an abstract design without specifying every detail, which can later be refined to more concrete implementations. We also require these abstract specifications to be executable and verifiable so that simulation and verification can be deployed in the early stages of hardware development. One problem in specifying a high-level design is the conflicting demands of making the specification abstract and simple, at the same time executable and verifiable.

In industry, the microarchitectural design of hardware is usually specified with natural languages. Such specifications are often ambiguous, and misinterpretation can introduce design faults in its implementation. Many attempts to specify high-level designs in programming languages such as C++ [SCV+00] or hardware description languages like VHDL [WFF00] have been made. These languages are popular, but their lack of formal semantics precludes formal verification.

Windley has proposed generic interpreters to represent a family of operations in microprocessors using higher-order functions [WFF00]. We can verify hardware by first proving the correctness of an abstract design with higher-order functions,

and then replace them with functions that represent concrete circuits. The objective of generic interpreters is to reduce the verification cost by eliminating repetitive proof obligations. However, generic interpreters cannot be executed without instantiating the higher-order functions.

Our goal is to define an abstract specification of hardware for simulation and verification, which can be also used as a reference model during the implementation of hardware [150]. We discuss it by examining the modeling techniques used in the FM9801 verification project [500]. This project verified a microprocessor model with various features found in today's microprocessors: speculative execution with branch prediction, out-of-order execution of instructions with multiple pipelined execution units, and precise exceptions. We mechanically verified that the entire FM9801 microarchitectural model implements its ISA model using the ACL2 theorem prover. Its proof script is publicly available [500].

In the FM9801 project, the high-level specification has been used both for simulation and verification. Using the ACL2 language, we defined the machine specification with next-state functions. ACL2 can execute these next-state functions, and also prove properties about them. We made use of the ACL2 encapsulation mechanism for modeling, so that the specification can be abstract and executable.

This paper is organized as follows. First, we discuss function encapsulation in the ACL2 logic in Section 2. Then we discuss two examples from the FM9801 modeling in Section 3: a branch predictor in Subsection 3.1 and a pipelined multiplier in Subsection 3.2. Then, we conclude in Section 4.

2 ACL2 and Function Encapsulation

In the ACL2 logic [500, 500], commands that modify the theorem prover's database are called *events*. A function definition is an event. For instance, the following `defun` event defines function `succ` which returns its argument plus one.

```
(defun succ (x) (+ x 1))
```

Theorems are introduced with `defthm` events. The following theorem states that, if `x` is an integer, `(succ x)` is also an integer.

```
(defthm succ-is-integer
  (implies (integerp x) (integerp (succ x))))
```

The ACL2 prover can prove this simple theorem automatically. For difficult theorems, the user has to guide the prover by supplying hints or breaking down the proof.

ACL2 encapsulation is a mechanism to introduce abstract functions with constraints. For instance, we can define a mathematical group with the encapsulation mechanism. A group can be represented by a 4-tuple `(in-group, prod, unit, inv)`, where `(in-group x)` is a membership predicate for the group, `(prod x y)` returns the product of `x` and `y`, `(unit)` is the unit element, and `(inv x)` returns the inverse of `x`. A concrete example of such an abstract group is

```

(encapsulate ((in-group (x) t)
              (prod      (x y) t)
              (unit      ()   t)
              (inv        (x)  t))
  (local (defun in-group (x) (integerp x)))
  (local (defun prod      (x y) (+ x y)))
  (local (defun inv        (x)  (- x)))
  (local (defun unit      ()    0))

  (defthm closure
    (and (implies (and (in-group x) (in-group y))
                  (in-group (prod x y)))
         (implies (in-group x)
                  (in-group (inv x)))))

  (defthm identity
    (and (implies (in-group x)
                  (equal (prod x (unit)) x))
         (implies (in-group x)
                  (equal (prod (unit) x) x))))

  (defthm inverse
    (and (implies (in-group x)
                  (equal (prod x (inv x)) (unit)))
         (implies (in-group x)
                  (equal (prod (inv x) x) (unit)))))

  (defthm associativity
    (equal (prod (prod x y) z) (prod x (prod y z))))
) ; end of encapsulate

(defthm inv-cancellation-1
  (implies (and (in-group x) (in-group y))
    (equal (prod (inv x) (prod x y)) y))
  :hints (("goal" :in-theory (disable associativity)
              :use (:instance associativity
                          (x (inv x)) (y x) (z y)))))

(defthm inv-cancellation-2
  (implies (and (in-group x) (in-group y))
    (equal (prod x (prod (inv x) y)) y))
  :hints (("goal" :in-theory (disable associativity)
              :use (:instance associativity
                          (x x) (y (inv x)) (z y)))))

(defthm inverse-of-product
  (implies (and (in-group x) (in-group y))
    (equal (inv (prod x y))
          (prod (inv y) (inv x))))
  :hints (("goal" :use (:instance inv-cancellation-1
                          (x (prod x y))
                          (y (prod (inv y) (inv x)))))
))

```

Fig. 1. ACL2 definition of an abstract group with encapsulation. Expressions following “:hints” are proof hints to the ACL2 prover.

(**integerp**,+,0,-), i.e., the set of integers with arithmetic summation as its product operation.

Figure 1 shows the definition of the abstract group with encapsulation. ACL2 **encapsulate** has the syntax of (**encapsulate** ($sig_1 \dots sig_n$) $ev_1 \dots ev_m$), where sig_i is the signature of an introduced function, and ev_i is an event such as **defun** and **defthm**.

A signature defines the arity of a function. Signature (**in-group** (x) t) introduces **in-group** as a function that takes one argument and returns one value. Similarly, functions **prod**, **unit** and **inv** take two arguments, zero arguments and one argument, respectively.

Events in ACL2 encapsulation can be either *local* or *exported*. Event ev appearing in a form (**local** ev) is local. Otherwise, the event is exported from the **encapsulate** form. Local events have similar scopes as private objects in C++ classes. The functions or theorems resulting from local events are visible only inside the **encapsulate** form. Exported events are similar to public objects in C++. They are visible from both inside and outside of the **encapsulate** form.

In Fig. 1, we locally define **in-group**, **prod**, **unit**, and **inv** to be **integerp**, **+**, 0, and **-**, respectively. With these definitions, the exported theorems **closure**, **identity**, **inverse**, and **associativity** are trivially true. These theorems correspond to the fundamental axioms of mathematical groups.

From outside of the **encapsulate** form, however, the group (**in-group**, **prod**, **unit**, **inv**) is viewed as the abstract group that only satisfies the exported four theorems. These theorems are *constraints* of the introduced functions. Local function definitions are required only to establish the consistency of the prover's theory with the abstract functions. Because the local functions are *witnesses* that satisfy the exported theorems, the extended theory cannot be inconsistent.

Outside of the **encapsulate** form, we can prove theorems about the constrained functions using only the exported theorems. In Fig. 1, we proved three theorems whose mathematical notations are $x^{-1} \cdot (x \cdot y) = y$, $x \cdot (x^{-1} \cdot y) = y$, and $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$. These theorems are provable from the four exported theorems, and they are true for any mathematical groups. However, associativity $x \cdot y = y \cdot x$ cannot be proven outside the **encapsulate**. In fact, associativity does not hold for arbitrary groups, even though it is true for the locally defined group (**integerp**,+,0,-).

3 Hardware Modeling with Encapsulation

We use ACL2 encapsulation to define a behavioral specification of an abstract design. In our approach, exported theorems are used for verification, while locally defined functions are used for simulation. In the current implementation of ACL2, locally defined functions are not executable. This is because local functions are used only to guarantee the consistency of the extended theory. We define a new macro **encapsulate-impl**, which treats locally defined functions as if they were exported functions only when simulation is performed. During the verification, **encapsulate-impl** hides all local events.

We examine two examples used in the FM9801 project. The first example models the branch predictor without specifying a concrete prediction algorithm. The other example models a pipelined multiplier.

3.1 Modeling of Branch Predictors

The branch predictor in the FM9801 returns 1 when a conditional branch is likely to be taken, and 0 when it is unlikely. This result is used to control the speculative execution of instructions before the branch direction is determined.

A number of branch prediction algorithms are known, and we may not want to decide which algorithm should be used when specifying a high-level design. For verification purposes, we modeled the branch predictor to be a component that nondeterministically returns 1 and 0. If we verify that a microprocessor design always executes the program correctly with this model, it is guaranteed to work with any branch predictor.

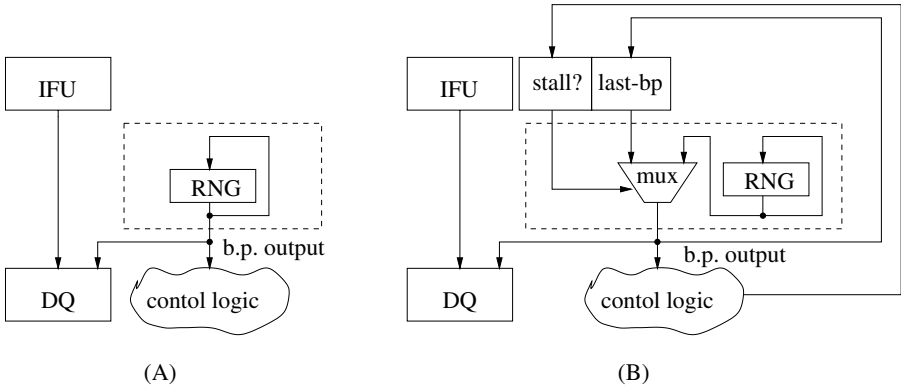


Fig. 2. Models of branch predictors. Branch prediction is performed on the instruction in the instruction fetch unit (IFU), and its result is stored in the dispatch queue (DQ).

For execution purposes, we can locally define the branch predictor with a random number generator (RNG). We defined a function (`rand n seed`) that generates a random natural number less than `n` using `seed`, and (`gensseed seed`) that calculates the next seed.

Using ACL2 encapsulation, we define the nondeterministic branch predictor as follows:

```

(encapsulate-impl ((branch-predict? (MA) t)
                  (step-BP-seed (MA) t))

  (local
    (defun branch-predict? (MA)
      (rand 2 (BP-seed (MA-BP MA)))))

  (local
    (defun step-BP-seed (MA)
      (genspeed (BP-seed (MA-BP MA)))))

  (defthm bitp-branch-predict? (bitp (branch-predict? MA)))
) ;end of encapsulate-impl

```

The function call `(branch-predict? MA)` takes processor state `MA` and returns the output value from the branch predictor. With the local definition, it generates a random number from the seed stored in our branch predictor model. Function `MA-BP` returns the branch predictor state in the processor state and `BP-seed` returns the seed in the predictor. Function `step-BP-seed` calculates the next-state of the random number generator. Then the exported theorem `bitp-branch-predict?` states that `branch-predict?` returns a bit (1 or 0). From the verification perspective, we have defined an arbitrary branch predictor that satisfies only this theorem. Figure 4.1 (A) shows the implementation. In this figure, encapsulation hides the part surrounded by a dashed box during the verification.

We can replace our local definition of `branch-predict?` and the same abstract model is defined as long as the exported theorem `bitp-branch-predict?` is true. We actually tried different versions of branch predictors in the FM9801 model and ran simulations to detect bugs in the early FM9801 design.

Theorem `bitp-branch-predict?` is the only constraint about the branch predictor that can be used during the verification of the entire FM9801 microarchitecture. However, it turns out that our early design of the FM9801 does not always work correctly with this simple branch predictor model. The FM9801 performs more than one branch prediction for the same branch instruction if this branch instruction stalls at the instruction fetch unit. If the results from repeated predictions differ, an early version of FM9801 starts the instruction fetch from an incorrect address.

There are two approaches to the problem. In the first approach, we consider this as a bug of the design, fix it, and verified the new design. We presented the verification of FM9801 using this approach in [Savarese]. In the second approach, we add a constraint to the branch predictor so that it returns the same result for

¹ The number of variables to store the branch history varies depending on the prediction algorithm. We consider the seed variable represents the collection of these variables. It is just an integer during the simulation, but the fact that it is an integer is not exported. From the verification perspective, the seed variable can be considered to have much richer structure, to which `step-BP-seed` selectively stores the history information.

the same instruction when predictions are repeated. In the rest of this section, we explain how we can model such a predictor using encapsulation.

The new branch predictor model is depicted in Figure 1(B). First, we add two auxiliary variables **stall?** and **last-bp** to the machine state. Variable **stall?** is set whenever an instruction stalls at the instruction fetch unit. Variable **last-bp** stores the output from the branch prediction in the previous cycle. The following **encapsulate** form defines the new branch predictor with the additional constraints.

```
(encapsulate-impl ((branch-predict? (MA) t)
                  (step-bp-seed (MA) t))
  (local
    (defun branch-predict? (MA)
      (b-if (BP-stall? (MA-BP MA))
        (BP-last-bp (MA-BP MA))
        (rand 2 (BP-seed (MA-BP MA))))))
    (local
      (defun step-bp-seed (MA)
        (gensed (BP-seed (MA-BP MA))))))

  (defthm bitp-branch-predict? (bitp (branch-predict? MA)))



  (defthm repeat-branch-prediction
    (implies (b1p (BP-stall? (MA-BP MA)))
      (equal (branch-predict? MA)
        (BP-last-bp (MA-BP MA)))))
) ; end of encapsulate
```

Our local definition of the branch predictor checks variable **stall?**, and returns the last branch prediction result **last-bp** if it is 1. Otherwise, it returns a random value. The exported theorems prove that the type of the branch predictor output is a bit, and that the predictor returns the same result as the previous prediction when a branch instruction stalls. Any branch predictor whose results are the same for repeated predictions for a particular instruction is a refinement of this abstract model.

It turned out that these two exported theorems are all we needed to verify the early FM9801 design. The ACL2 prover mechanically checked the correctness of the design with the new branch predictor model.

² The result of **branch-predict?** depends on the auxiliary variables **stall?** and **last-bp**, and this dependency is not completely hidden by encapsulation. A cleaner model can be given by adding copies of these variables inside the predictor model, encapsulating the updating functions, and separating the auxiliary variable from the machine state **MA**.

3.2 Modeling of a Pipelined Execution Unit

In this subsection, we show another example to discuss behavioral modeling of pipelined execution units. *Uninterpreted functions*  are often used to represent the data-path in processor verifications. By representing the outputs from execution units with expressions containing uninterpreted functions, we can verify the results of some processors by syntactic comparison . However, uninterpreted functions are not executable because they have no interpretations.

Using our approach with function encapsulation, we can run simulation using locally defined functions that calculate execution unit outputs, while we treat the functions as uninterpreted during verification. As we show later, we can also instantiate the abstract design with a detailed design which is separately proven to satisfy the function constraints.

We examine a three-stage pipelined multiplier in the FM9801. There are a number of ways to implement a pipelined multiplier, and we may not want to give a specific implementation when defining a high-level design. The specification only needs to satisfy the fact that the output from the last stage is the product of the operands given at the first stage. Using function encapsulation, we define the data-path of our multiplier as follows:


```
(encapsulate-impl ((ML1-output (a b) t)
                  (ML2-output (data) t)
                  (ML3-output (data) t))
  (local (defun ML1-output (a b)
            (cons a b)))

  (local (defun ML2-output (data)
            data))

  (local (defun ML3-output (data)
            (word (* (car data) (cdr data))))))

  (defthm ML3-output-correct
    (word-p (ML3-output data)))

  (defthm ML-output-correct
    (implies (and (word-p a) (word-p b))
      (equal (ML3-output (ML2-output (ML1-output a b)))
        (word (* a b)))))
) ; end of encapsulate
```

We introduce three constrained functions to define the output from each stage as depicted in Figure  (A). The first function `ML1-output` returns a pair of two input operands, the second stage function `ML2-output` returns its input as it is, and the third stage function calculates the product of a pair modulo 2^{16} . These three functions satisfy two constraints: the output has the 16-bit data word type and the composition of the three functions produces the product modulo 2^{16} of

the two input operands. Predicate `word-p` is a type predicate for 16-bit words, and function `word` defines the modulo operation.

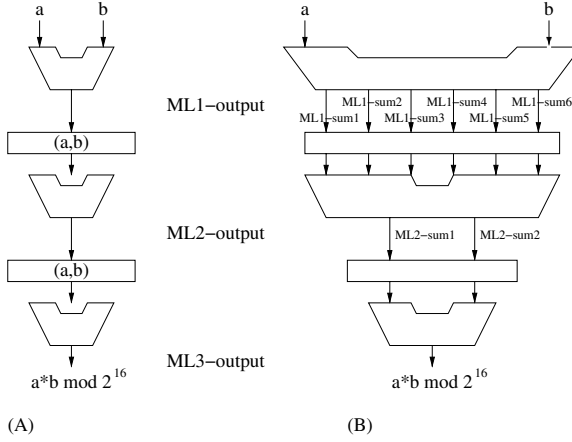


Fig. 3. Three-stage multiplier.

The control logic of the multiplier is given separately and is relatively complex, while the local definition of the data-path is very simple. Unlike using uninterpreted functions to represent the data-path, the model is executable, enabling us to simulate the entire processor model. The reader may wonder whether such a simple data-path specification is useful for simulation. We would like to stress that this simplicity is the strength of our abstract specifications. For the simulation and verification of the control logic and the rest of the processor design, it does not matter how the data-path is implemented. It is better to have a simpler model so that we can quickly write the specification and start debugging earlier.

Our model represents the data-path of an arbitrary three-stage multiplier, because the local definitions are hidden and exported theorems only imply basic facts about multipliers. Thus, the validity proof of the FM9801 processor using this model implies the validity of the processor using any multiplier implementations.

In order to show our point, we first verified the FM9801 model with this abstract multiplier, then replaced it with a separately verified implementation of the multiplier, and re-verified the combined model. We implemented a Wallace tree multiplier [Wallace] with 16 shifters, 14 carry-save adders and a ripple-carry adder. Our implementation of the multiplier has three stages as shown in Fig. 3 (B). The first stage takes two operands and produces 6 partial sums, the second stage reduces 6 partial sums into 2, and the last stage adds them to produce the final answer. In the following definition, functions $MLn\text{-}data$ for $n = 1, 2$ are pairing functions, and $MLn\text{-}data\text{-}sumk$ for $n = 1, k = 1, 2$ and

$n = 2, k = 1, 2, 3, 4, 5, 6$ are element accessors. The combinational circuits consisting of shifters and adders are represented by functions `ML n -sum k` , whose definitions are not given here.

```
(defun ML1-output (a b)
  (ML1-data (ML1-sum1 a b) (ML1-sum2 a b) (ML1-sum3 a b)
            (ML1-sum4 a b) (ML1-sum5 a b) (ML1-sum6 a b)))

(defun ML2-output (data)
  (ML2-data (ML2-sum1 (ML1-data-sum1 data) (ML1-data-sum2 data)
                    (ML1-data-sum3 data) (ML1-data-sum4 data)
                    (ML1-data-sum5 data) (ML1-data-sum6 data))
            (ML2-sum2 (ML1-data-sum1 data) (ML1-data-sum2 data)
                    (ML1-data-sum3 data) (ML1-data-sum4 data)
                    (ML1-data-sum5 data) (ML1-data-sum6 data))))

(defun ML3-output (data)
  (word (+ (ML2-data-sum1 data) (ML2-data-sum2 data))))
```

This concrete implementation of the multiplier satisfies the constraints of the abstract multiplier defined earlier. The output from the last stage satisfies the type condition. The second constraint theorem `ML-output-correct` is satisfied, because the ACL2 theorem prover can mechanically check the following theorem.

```
(defthm correctness-of-multiplier
  (implies (and (word-p a) (word-p b))
    (let ((s1 (ML1-sum1 a b)) (s2 (ML1-sum2 a b))
          (s3 (ML1-sum3 a b)) (s4 (ML1-sum4 a b))
          (s5 (ML1-sum5 a b)) (s6 (ML1-sum6 a b)))
      (let ((sum1 (ML2-sum1 s1 s2 s3 s4 s5 s6))
            (sum2 (ML2-sum2 s1 s2 s3 s4 s5 s6)))
        (equal (word (+ sum1 sum2))
                (word (* a b)))))))
```

We replaced the abstract multiplier definition with this implementation. By using the theorem `correctness-of-multiplier`, and the verification script originally used to verify the design with the abstract definition, the ACL2 prover successfully verified the entire proof script without additional human assistance.

4 Conclusion

We have presented abstract modeling techniques used in the FM9801 verification project. The main idea is to use function encapsulation to define local simple functions for fast simulation, and hiding them from the verification engines. For verification, we can use only the exported constraints of the abstract design. Any implementation satisfying these constraints is a refinement of the abstract

design, and the verification of the abstract design can be applied to the refined implementation.

Those who are familiar with the ACL2 theorem prover will notice that our approach can be mimicked by using the ACL2 `disable` feature, which temporarily hides function definitions from the prover engine. There are two merits of using the encapsulation mechanism. One is the proof security; unlike disabling functions, encapsulation never reveals the hidden definitions to the prover engine by human mistake. The other merit is the clarity of abstract specifications. An abstract specification using encapsulation clearly lists, as exported theorems, the properties that must be satisfied by its concrete implementation. It also distinguishes the code used for simulation from other code. This distinction is important if we use the abstract specification as a reference model when implementing its concrete circuit designs. In a sense, our approach provides the way to define structural specifications as opposed to traditional flat hardware specifications in ACL2.

The features provided by ACL2 were sufficient to perform this modeling exercise, but more sophisticated languages are desired. One major problem is the current ACL2 implementation of encapsulation does not allow the execution of local functions. We constructed a user-defined macro `encapsulate-impl` that made this possible.

Another problem with the ACL2 function encapsulation is its ability to hide only function specifications but not variables. In Subsection 3.2, we have seen that the functions representing the data-paths are encapsulated. However, the data-structure representing the pipeline latch states are not encapsulated together. As a result, the local model used for simulation cannot be completely hidden from the verification engine. We believe encapsulation mechanism for both data and functions are helpful when modeling state-holding devices.

Our modeling technique was critical to the successful verification of the FM9801, which is one of the most complex hardware designs ever completely verified. It allowed quick modeling of a processor design, its simulation and verification. Otherwise, we would have needed to implement the detail of the processor before simulation and verification.

Our modeling technique can be useful in industry, where microarchitectural specifications are desirable for early simulation and verification. When implementing its circuit design, the abstract design becomes the reference model without ambiguity, which must be refined rigorously.

Acknowledgement

We thank J Moore and Matt Kaufmann for implementing and supporting the ACL2 theorem prover. We also thank Damir Jamsek for reading our manuscript.

References

- BD94. Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer Verlag, 1994.
- HS99. Warren A. Hunt, Jr. and Jun Sawada. The FM9801 microprocessor verification. *IEEE Micro*, 19(3):47–55, May/June 1999.
- KM96. Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.
- KM99. Matt Kaufmann and J Strother Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual*. 1999. URL: <http://www.cs.utexas.edu/users/moore/acl2/acl2-996-nrm1-user's-manual>.
- KMM00. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.
- Saw99a. Jun Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, December 1999. Also available from <http://www.cs.utexas.edu/users/sawada/dissertation/diss.html>.
- Saw99b. Jun Sawada. Verification scripts for FM9801 pipelined microprocessor design, 1999. URL: <http://www.cs.utexas.edu/users/sawada/fm9801/>.
- SCV⁺99. Patrick Schaumont, Radim Cmar, Serge Vernalde, Marc Engels, and Ivo Bolsens. Hardware reuse at the behavioral level. In *Design Automation Conference (DAC '99)*, pages 552–557. ACM Press, June 1999.
- Sho84. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
- VB99. Miroslav N. Velev and Randal E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, (CHARME '99)*, volume 1703 of *LNCS*, pages 37–53. Springer Verlag, 1999.
- Wal64. C. E. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, February 1964.
- WH99. Dyson Wilkes and M.M. Kamal Hashmi. Application of high level interface-based design to telecommunications system hardware. In *Design Automation Conference (DAC '99)*, pages 552–557. ACM Press, June 1999.
- Win90. Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, jun 1990.

A Methodology for the Formal Analysis of Asynchronous Micropipelines

Antonio Cerone¹ and George J. Milne²

¹ Software Verification Research Centre
The University of Queensland, Brisbane QLD 4072, Australia
`antonio@it.uq.edu.au`

² Advanced Computing Research Centre
University of South Australia, Adelaide SA 5095, Australia
`milne@cs.unisa.edu.au`

Abstract. In this paper we present a process algebra approach for the integrated verification of correctness and performance in concurrent systems. The verification procedure is entirely performed within the Circal process algebra, without any recourse to other formalisms. Performance is characterised in terms of logical properties, which do not incorporate explicit time. Such properties are then interpreted in terms of degree of parallelism and allow the quantitative evaluation of the throughput of the system. The approach has been applied to two four-phase handshaking protocols, which are motivated by the implementation of the AMULET2 asynchronous RISC processor. Both correctness and performance properties are captured in the same verification framework and automatically proved using the Circal System.

1 Introduction

Formal methods have tended to concentrate on verification of correctness of software and/or hardware systems. Typical practical reasons to apply formal methods have been to ensure the safety of a software or hardware system. Once the safety properties that the system is required to meet are specified, verification consists in checking that the software and/or hardware implementation of the system is equivalent to its specification. However, in the broadest sense verification must include a performance analysis of the system and formal methods should be extended to allow performance to be included in the analysis. This is especially the case for those application domains, such as hardware systems, where performance plays a key rôle in the choice between alternative technologies.

Past techniques in performance analysis have featured model development based on experience and intuition and analysis by simulation or analytical techniques based on assumptions that are known to be approximate. Two major modelling approaches that have been used for both verification and performance analysis are timed Petri Nets and timed process algebras [2]. A standard method to introduce performance analysis into these two formal methods is to associate

time with the actions of the process algebra and the transitions (or equivalently the residence time of the places) of the Petri net. These times could be either deterministic or stochastic. In the analysis of the former we can use max-plus algebra [10]. The latter leads to stochastic Petri nets and stochastic process algebras. Performance analysis is then based on a derived Markov chain.

Adding time in this way to both Petri net and process algebra models increases the complexity of any analysis procedure as compared with untimed case. Markov chain analysis is particularly restricted by state explosion. Some Petri net methods allow abstraction of time from the model where it is not significant by having zero time transitions but most stochastic process algebras do not allow abstraction with zero time transitions.

In a previous paper [11] we have introduced an approach to the integrated verification of correctness, timing and performance properties in concurrent systems, using the Circal process algebra and its mechanisation, the Circal System [12]. Our approach does not make any explicit use of time. Performance is not determined by absolute values, but by the degree of parallelism of the system components.

In this paper we develop our approach [11] into a rigorous methodology and we apply it to two different four-phase handshaking protocols, proving both their correctness and interesting performance properties, which allow a quantitative evaluation of the throughput of the system.

2 The Process Algebra

Process algebras are mathematical formalisms for describing systems of interacting Finite State Machines (FSMs). The interaction is given by synchronising the transitions that occur in different FSMs. This can be done in several ways, which differentiate the many process algebras that appear in the literature [13, 14, 15, 16].

2.1 Hierarchy of Processes

Every process algebra has one (or more) *parallel composition operator*, a *hiding operator* and a *relabelling operator*. The combination of the three operators allows for the structure of any system to be modelled as a hierarchy of abstraction levels, as shown in Figure 1(a). Every box represents a process and is decomposed into the components at the lower level that it is connected to. For example, process $S_{1,2}$ consists of two components: process $S_{2,2}$ and process $S_{2,3}$. Only the boxes that are leaves of the hierarchy, not necessarily at the lowest level, explicitly encapsulate behaviours ($S_{1,3}$, $S_{2,2}$, $S_{3,1}$, $S_{3,2}$ and $S_{3,3}$). In the following we will call such leaves *behavioural processes*. Every process interacts with other processes through communication ports. Interaction between processes occurs through the actions that are associated with the ports. In Circal [12], CSP [17] and LOTOS [18], a communication channel connects all the ports that are labelled with the same action. In CCS [19], actions are coupled in complementary pairs (input and output actions) and a directed communication channel connects the

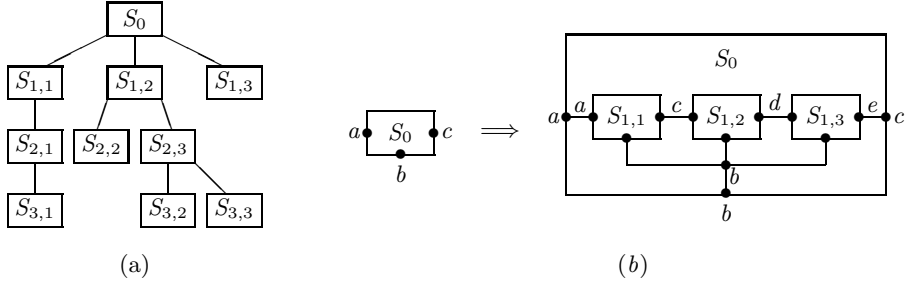


Fig. 1. (a) Hierarchy of system components; (b) Graphical representation of parallel composition with hiding and relabelling.

two ports that are labelled with complementary actions. In our approach we adopt the communication paradigm of Circa, CSP and LOTOS. Rather than formally introducing one of these process algebraic languages, we will present all necessary concepts in an intuitive graphical fashion.

If we look at the hierarchy given in Figure 1(a) from a bottom-up point of view, every process, apart from the root, is embedded within the parent process by composing it in parallel with other processes; by hiding some of the actions that are used for interaction; and possibly by relabelling other actions. For example, $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ are embedded within S_0 as shown in Figure 1(b). Communication ports are represented as bullets on the outline of the box. Communication channels are represented by lines connecting two ports, when only two ports are involved in the communication, or by a bullet connected with all the ports involved in the communication, when the involved ports are more than two. The action associated with a port is written next to the port, if that port is not involved in any communication at the given abstraction level, next to the bullet or to any line describing the communication in which the given port is involved, otherwise. The embedding of a set of processes within the next abstraction level is represented by a box surrounding the set of processes, with new bullets (with the corresponding actions, which may be relabelled, written next to them) on its boundary to represent all the ports that are not hidden after the composition. These new bullets are connected by lines to any of the corresponding internal bullets. For example, in Figure 1(b), $S_{1,1}$ and $S_{1,2}$ communicate through the channel labelled by action c , $S_{1,2}$ and $S_{1,3}$ through the channel labelled by action d and $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ through the three-way channel labelled by action b . Actions c and d are hidden in S_0 ; a and b are visible at the next abstraction level as ports of S_0 ; e is relabelled with c as a port of S_0 .

In this way, the box that embeds a set of processes represents the *interface* of the composite process. Every process has a *sort*, which is the set of action names which label the ports on the box that embeds its components. For example, in Figure 1(b), S_0 and $S_{1,1}$ have sort $\{a, b, c\}$, $S_{1,2}$ has sort $\{b, c, d\}$ and $S_{1,3}$ has sort $\{b, d, e\}$. For a behavioural process, its sort (or interface) must contain at least all the actions that occur in the embedded behaviour.

2.2 Process Behaviour

The parallel composition of behavioural processes may be expanded into a global

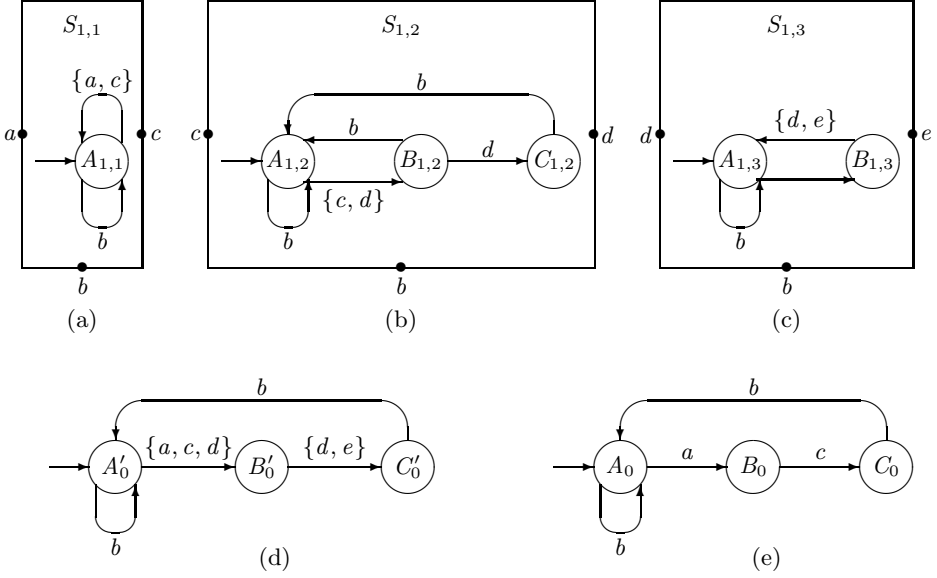


Fig. 2. (a–c) Interfaces and behaviours of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$; (d) Behaviour of the parallel composition of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$; (e) Behaviour of S_0 .

behaviour. Every state of the global behaviour is given by the product of component states, one for each component. The precise semantics of parallel composition depends on the specific process algebra involved. In this paper we consider the approach adopted by the Circal process calculus [15], where every transition between states is labelled with a (possibly empty) set of actions.

Given a set of processes and a set of transitions, one for each process, the transitions of the set *may synchronise* if and only if, for each action that belongs to the label of at least one transition, if the action does not occur in the label of a transition of the given set, then it does not belong to the sort of the process to which such a transition belongs; here causally independent actions are synchronised. If the transitions of the set *must synchronise* if and only if there is at least one action in the intersection of their labels; here identical actions from distinct component processes synchronise. When transitions of different processes synchronise, the label of the transition of the composite process is the union of the labels of all components.

For instance, if we compose $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ in Figure 2(a–c), then the transition labelled by $\{a, c\}$ from $A_{1,1}$ to $A_{1,1}$ in $S_{1,1}$, the transition labelled by

$\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$, and the transition labelled by $\{d\}$ from $A_{1,3}$ to $B_{1,3}$ in $S_{1,3}$, may synchronise. The corresponding transition of the composite process, which is represented in Figure 4(d) is from $A_{1,1} \times A_{1,2} \times A_{1,3}$ to $A_{1,1} \times B_{1,2} \times B_{1,3}$ and is labelled by $\{a, c, d\} = \{a, c\} \cup \{c, d\} \cup \{d\}$. Notice that in Figures 4(a-c) we have represented the sets that consist of a single action by writing just the action name without brackets.

The behaviour of the composition of $S_{1,1}$, $S_{1,2}$ and $S_{1,3}$ is given in Figure 4(d). Here: $A'_0 = A_{1,1} \times A_{1,2} \times A_{1,3}$, $B'_0 = A_{1,1} \times B_{1,2} \times B_{1,3}$ and $C'_0 = A_{1,1} \times C_{1,2} \times A_{1,3}$. Notice that the transition labelled by $\{a, c\}$ from $A_{1,1}$ to $A_{1,1}$ in $S_{1,1}$ must synchronise with the transition labelled by $\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$. Analogously, the transition labelled by $\{c, d\}$ from $A_{1,2}$ to $B_{1,2}$ in $S_{1,2}$ must synchronise with the transition labelled by $\{d\}$ from $A_{1,3}$ to $B_{1,3}$ in $S_{1,3}$.

After hiding c and d and relabelling e with c , we obtain the behaviour of S_0 , which is given in Figure 4(e).

2.3 Process Models of System Components

In our verification methodology we utilise the core modelling object, namely a *process*, to model four quite distinct artifacts. The first of these is when we model the *physical components* of the system under investigation.

The second artifact that we model by a process, or processes, are *assumptions* on the behaviour of the system. These assumptions usually relate to context or environmental restrictions and generally simplify the system behaviour when composed with it using the composition operator.

The third artifact that is modelled by a process is the specific *property* which we want to verify as holding in the system and which is used to describe the notion of system correctness.

The fourth artifact that is also modelled by a process is a *refinement* of part of the behaviour of another process. By composing a given process with a refinement process we extend the behaviour of the given process. Together with the hiding operator, this allows the definition of a new *view* of the system. In this paper we will consider only a single type of refinement, namely a *time interval refinement*. Processes may also model different artifacts at the same time. We will see processes that are both refinements and assumptions.

3 Modelling Asynchronous Micropipelines

In a RISC processor the instruction pipeline is composed of logic stages and latches. Progress through a synchronous pipeline is managed by the clock; once the logic has completed evaluation all the latches are clocked at the same time simultaneously moving all the instructions to the next pipeline stage. In an asynchronous micropipelined processor [10] the evaluation of a pipeline stage is governed by local interactions with its neighbours using a request acknowledge handshaking protocol. It is possible that one stage is evaluating while at the same time a stage further on is transferring an instruction to its neighbour. Thus,

whilst the performance of a synchronous pipeline is governed entirely by the clock rate that can be achieved with a particular logic design, the performance of an asynchronous pipeline depends as well on the design of the handshaking controls for each stage. In particular, if the asynchronous logic pipeline is to be as fast as the synchronous one it must be possible for all the evaluation of logic stages to overlap as in the synchronous case.

In our example, we analyse the correctness and the performance of two micropipelines which are motivated by the implementation of the AMULET2 asynchronous RISC processor [4]. Their control logics consist of a four-phase handshake protocol where the rising signals are active. The datapaths consist of just sequences of latches without logic stages in between. The whole micropipelines can then be seen as FIFO queues.

3.1 Specification

The specifications of the single stages of the two latch control circuits we are going to model are given by the STGs (Signal Transition Graphs) [1] in Figure 3 where:

- the dashed arrows denote the orderings that must be maintained by the environment (*assumptions* about the environment);
- the solid arrows denote the orderings that must be ensured by the circuit itself (*properties* of the circuit);
- a solid circle is attached to an arrow in order to denote that the target of such an arrow is initially enabled to occur.

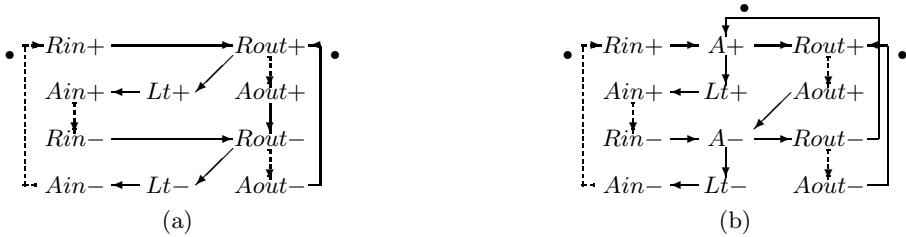


Fig. 3. (a) STG of simple four-phase latch control circuit; (b) STG of semi-decoupled four-phase latch control circuit.

Rin and $Rout$ define the input and output *request signals*. Ain and $Aout$ define the input and output *acknowledgement signals*. The Lt latch control signal causes the data latch to be open when low ($Lt-$) and closed when high ($Lt+$). The STGs in Figure 3 show that when input data is available ($Rin+$) the latch may close ($Lt+$) and then the input may be acknowledged ($Ain+$); when the output data has been acknowledged ($Aout+$) the latch may open again ($Lt-$). It is responsibility of the environment to ensure that an input acknowledgement signal ($Ain+$) will eventually reset the input request ($Rin-$), the reset of the input

acknowledgement ($Ain-$) will be eventually followed by a new input request signal ($Rin+$), an output request signal ($Rout+$) will be eventually acknowledged ($Aout+$), the reset of the output request ($Rout-$) will be eventually followed by the reset of the output acknowledgement ($Aout-$). These assumptions about the environment are denoted by dashed arrows in Figure 1. Notice that the assumptions are the same for both STGs.

3.2 Implementation

The STGs in Figure 1 can be implemented into circuits using informal or semi-formal synthesis techniques. A possible implementation of the STG in Figure 1(a) is given in Figure 4(a) and a possible implementation of the STG in Figure 1(b)

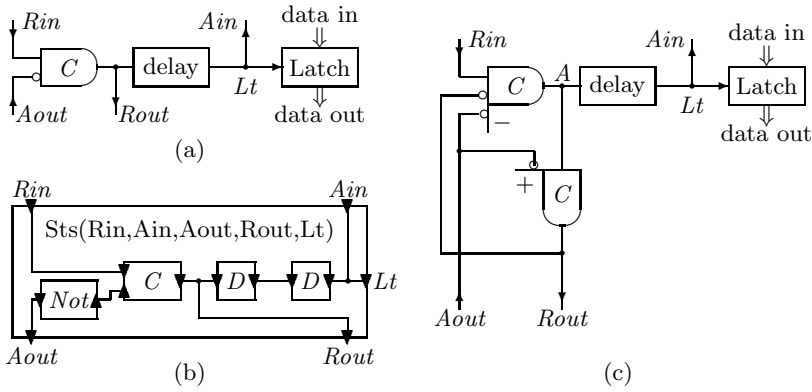


Fig. 4. (a) Simple four-phase latch control circuit; (b) Process algebra model of the simple four-phase latch control circuit; (c) Semi-decoupled four-phase latch control circuit.

is given in Figure 1(c) 1. In the implementation in Figure 4(a) we have used a conventional Muller C-gate, while in Figure 1(c) we have used two asymmetric Muller C-gates. In the notation for the asymmetric C-gates an input connected to the main body of the gate controls both edges of the output; an input connected to the extension marked with “+” controls only the rising edge; an input connected to the extension marked with “-” controls only the falling edge. A logic description of the C-gates in Figure 1(c) is given by the equations:

$$\begin{aligned} A &= Rin \cdot \overline{Rout} + A \cdot (Rin + \overline{Rout} + \overline{Aout}) \\ Rout &= A \cdot \overline{Aout} + Rout \cdot A \end{aligned}$$

The behaviours of the gates may be directly defined in the process algebra 1, 2. An alternative approach is to build gates at the CMOS level, by defining transistors directly as behaviours 3.

The Circal code for the asynchronous micropipeline example is available via ftp 4. In this paper we want just to give the flavour of what a Circal definition

looks like, by giving the Circal code that models a conventional C-gate, directly in the process algebra

```

Process C(Bool a, b, x, int n){
  Process S[6]
  S[0] <- a.1 S[2] + b.1 S[1] + (a.1 b.1 x.1) S[3]
  S[1] <- (a.1 x.1) S[3] + b.0 S[0] + (a.1 b.0) S[2]
  S[2] <- a.0 S[0] + (b.1 x.1) S[3] + (a.0 b.1) S[1]
  S[3] <- a.0 S[4] + b.0 S[5] + (a.0 b.0 x.0) S[0]
  S[4] <- a.1 S[3] + (b.0 x.0) S[0] + (a.1 b.0) S[5]
  S[5] <- (a.0 x.0) S[0] + b.1 S[3] + (a.0 b.1) S[4]
  return S[n]
}

```

and at the CMOS level

```

Process C(Bool a, b, x, int ai, bi){
  Bool h[3], l[3]
  Process S
  S <- PMos(a,h[0]) * PMos(b,h[1]) * Sequential(h[0],h[1],h[2]) *
    NMos(a,l[0]) * NMos(b,l[1]) * Sequential(l[0],l[1],l[2]) *
    ROutput(h[2],l[2],x,ai*bi) *
    return ~(S - (*l) (*h))
}

```

Once the gates have been defined as the basic components, the whole circuit is specified by composing in parallel the processes that define the gates, with additional processes to define the delays in the gates or on the wires. For example, the circuit in Figure 1(a) is represented in the process algebra by the *Sts* process defined in Figure 1(b). To increase readability we have not indicated the labels of the internal actions that are hidden in *Sts*. A similar representation as a process, say *Std*, may be given to the circuit in Figure 1(c). The solid triangles denote the ports that have boolean values. Such ports can be low, that is ready to generate a rising signal (+) or high, that is ready to generate a falling signal (−). For example, *Rin* can be ready to generate *Rin*− or *Rin*+. A port that is initially high is denoted by ▲; a port that is initially low is denoted by ▼. The two Circal processes given above are to alternative models for C. More information on how to model *Not* and *C* has been presented previously [10]. The *D* process, which has an input port on the left and an output port on the right, may be modelled by the behaviour in Figure 1(a), where the ports are initialised to the low level. *D*(*in*, *out*) defines an arbitrary delay between the *in* and *out* signals. In fact, an arbitrary number of external actions may occur between input *in*+ and output *out*+ or between input *in*− and output *out*−.

3.3 Correctness Verification

The two STGs defined in Figure 2 can be modelled in our process algebra framework by defining a process for every single relationship between pairs of signals

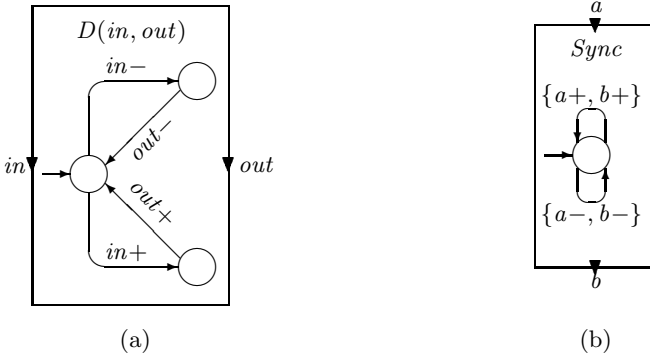


Fig. 5. (a) Delay process $D(in, out)$; (b) Synchronisation process $Sync$.

connected by an arrow, and by then composing all these processes together. An arrow between signals pre and $post$ without a solid circle attached is defined by the AC process represented in Figure 4(a). Signal $post$ is not initially enabled. Therefore, it must always occur either simultaneously or after an occurrence of signal pre . An arrow between signals pre and $post$ with the solid circle attached is defined by the IC process represented in Figure 4(b). Signal $post$ is initially enabled. Therefore, after the first occurrence of $post$ every further occurrence of $post$ must always occur either simultaneously or after an occurrence of sig-

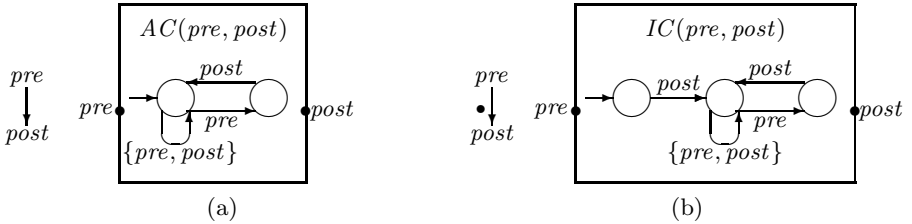


Fig. 6. (a) Process algebra model of an STG arrow without solid circle; (b) Process algebra model of an STG arrow with a solid circle attached.

nal pre . The dashed arrows in Figure 4 are represented in terms of processes in the same way as the solid arrows. However, these processes play different rôles in the specification in that processes that correspond to dashed arrows are *assumptions*, whereas processes that correspond to solid arrows are *properties*. The properties expressed by the STGs above are *safety* properties; they assert that, for all possible executions, the defined ordering of signals must not be violated. A verification methodology used in a process algebra framework consists of checking whether or not the process P that represents the safety property to be verified constrains the process S that represents the system. If P constrains S , then the property represented by P is not implicitly modelled in S ; on the

other hand if P does not constrain S , then the property represented by P is implicitly modelled in S , that is, the system satisfies the property.

The key point of the methodology is how to check whether or not one process constrains another. This can be done by an appropriate combination of the parallel composition and the equivalence checking procedure, which is available in the Circal System [15], a proof toolset which mechanises the Circal process algebra. In order to constrain a process S with another process P , we just need to compose S and P in parallel. Thus, if the constraint expressed by P is already implicitly modelled in S , then the parallel composition of S and P must be equivalent to S itself. If we denote the parallel composition by $*$ and the equivalence checking by \cong , we need to check the equivalence:

$$S * P \cong S \quad (1)$$

When the safety property only holds under the assumptions defined by a process A , equivalence (1) becomes

$$A * S * P \cong A * S \quad (2)$$

In this case the property is verified for the constrained system $A * S$. In (2) A can be any assumption, included a timing constraint, and P any property, included a performance property. In this way the verification schema given by (2) integrates correctness, timing and performance verification [15]. Therefore, properties are expressed in the same formalism as the model, as seen in formalisms based on propositional, first order logic and higher order logic [15].

In order to verify the correctness of the circuit given in Figure 1(a), which is modelled in Circal by the Sts process defined in Figure 1(b), we have to define A by composing in parallel the instantiations of AC and IC that represent the assumptions, and P by composing in parallel the instantiations of AC and IC that represent the properties. From the STG in Figure 1(a):

$$\begin{aligned} A &= IC(Ain-, Rin+) * AC(Ain+, Rin-)* \\ &\quad AC(Rout+, Aout+) * AC(Rout-, Aout-) \\ P &= IC(Aout-, Rout+) * AC(Aout+, Rout-)* \\ &\quad AC(Rin+, Rout+) * AC(Rin-, Rout-)* \\ &\quad AC(Rout+, Lt+) * AC(Rout-, Lt-)* \\ &\quad AC(Lt+, Ain+) * AC(Lt-, Ain-) \end{aligned}$$

If in (2) we replace S by the Sts process we can automatically verify using the Circal System [15] that the equivalence is true. That is the single stage modelled by $Sts(Rin, Ain, Aout, Rout, Lt)$ meets the properties modelled by P under the assumptions modelled by A . Therefore, the implementation given in Figure 1(a) is correct with respect to the specification given in Figure 1(a).

In order to verify the correctness of the circuit given in Figure 1(c), we have to define A and P from the STG in Figure 1(b). We can notice that A is the

same as before and

$$\begin{aligned}
 P = & IC(Aout-, Rout+) * IC(Rout-, A+) * AC(Aout+, A-)* \\
 & AC(Rin+, A+) * AC(Rin-, A-)* \\
 & AC(A+, Rout+) * AC(A-, Rout-)* \\
 & AC(A+, Lt+) * AC(A-, Lt-)* \\
 & AC(Lt+, Ain+) * AC(Lt-, Ain-)
 \end{aligned}$$

Analogously, we can automatically verify using the Circal System that the implementation given in Figure 4(c) is correct with respect to the specification given in Figure 4(b).

3.4 Performance Analysis

In the previous section we have seen how to automatically verify that the circuits defined in Figure 4 operate correctly. Whilst they both are correct with respect to the corresponding STG specifications, they show different performances.

Several stages of our asynchronous micropipeline controller may be connected in series as shown in Figure 7. Here St_i , $i = 1, 2, 3$, must be instantiated by

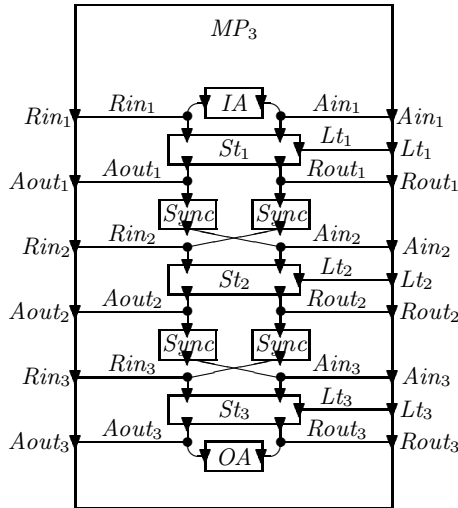


Fig. 7. A 3-stage micropipeline controller MP_3 .

$Sts(Rin_i, Ain_i, Aout_i, Rout_i, Lt_i)$, when using the control circuit given in Figure 4(a), by $Std(Rin_i, Ain_i, Aout_i, Rout_i, A_i, Lt_i)$ with A_i hidden, when using the control circuit given in Figure 4(c).

The latches corresponding to the controller form a FIFO of registers. The maximum potential parallelism for such a FIFO occurs when all the latches are

full at the same time. Whether or not such a potential parallelism is effectively attained depends on the handshake control protocol.

We define *throughput* the number of data items that can be passed through the pipeline per complete *handshake cycle*. This definition is justified by the practical observation that asynchronous pipelines are limited by the elapsed time for one handshake cycle in the control stages. A handshake cycle for the i -th stage is the sequence of events from Rin_i+ to Ain_i- . Previous studies have shown that there is a direct relationship between this throughput and the number of pipeline stages that can be full at a particular time [14].

We can notice that in the STG in Figure 1(a) $Aout_i$ must be low (and therefore the next latch empty — refer to Figure 1) before Lt_i can go high (and this latch become full). This is not the case for the STG in Figure 1(b), where the input side and the output side of the latch control stage are partly decoupled. A consequence of this decoupling is that a falling signal $Aout_i-$ that acknowledges the falling signal $Rout_i-$ in a given handshake cycle is concurrent with a rising signal Lt_i+ in the next handshake cycle. In this section we analyse the implication of this decoupling on the performance of the micropipeline.

The stages of the micropipeline are connected together as in Figure 1. The *Sync* processes, defined in Figure 1(b), have the purpose of synchronising the outputs of a stage with the corresponding inputs of the next stage, that is $Aout_i$ with Rin_{i+1} and $Rout_i$ with Rin_{i+1} . The *IA* and *OA* processes define the assumption on the input and on the output, respectively, that is:

$$IA = IC(Ain_1-, Rin_1+) * AC(Ain_1+, Rin_1-)$$

$$OA = AC(Rout_3+, Aout_3+) * AC(Rout_3-, Aout_3-)$$

In a previous paper [14] we have shown how to automatically verify that the input assumptions on the first stage and the output assumptions on the last stage imply inputs and outputs assumptions on the intermediate stages.

We can carry out the automated analysis of the performance of the micropipeline using the same methodology we have used in the previous section for the correctness proof [14]. Now, the safety property to be used will catch some performance aspects of the system rather than just orderings of event occurrences. We want to express the performance in terms of throughput. We then need a way to characterise when a stage of the micropipeline is full. The i -th stage starts to be full when Lt_i goes high. At this point the data in the corresponding buffer is latched. It cannot be overwritten by data coming from the $(i - 1)$ -th stage and is propagated to the $(i + 1)$ -th stage. When this data is latched in the $(i + 1)$ -th stage, Ain_{i+1} goes high and, as a result, $Aout_i$ goes high. At this point, since the current data is latched in the $(i + 1)$ -th stage, the buffer of the i -th stage can be overwritten. Therefore, the time interval where the i -th stage is full starts at Lt_i+ and ends at $Aout_i+$.

We introduce the $MI_1(from, to, mark)$ process given in Figure 1(a) to mark with the new *mark* abstract action the time interval between action *from* and action *to*. Action *mark* is called abstract because it does not belong to the set of physical actions performed by the system that we are modelling. When an

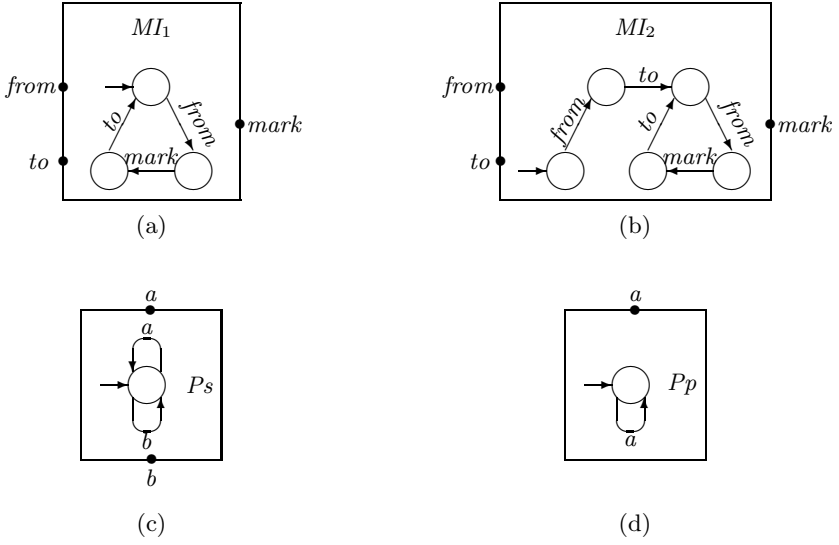


Fig. 8. (a) Interval refinement process MI_1 ; (b) Interval refinement process MI_2 ; (c) Property process Ps ; (d) Property process Pp .

instantiation of MI_1 is composed with a stage of the micropipeline as shown in Figure 4, we obtain a *time interval refinement* of the micropipeline. Action $full_i$ models a generic point in the time interval between Lt_i+ and $Aout_i+$. Notice that $MI_1(Lt_i+, full_i, Aout_i+)$ is receptive only to rising edges of Lt_i and $Aout_i$, that is only to actions Lt_i+ and $Aout_i+$. This is indicated by writing $+$ next to the ports of $MI_1(Lt_i+, full_i, Aout_i+)$ that are labelled by Lt_i and $Aout_i$.

Also notice that $MI_1(Lt_i+, full_i, Aout_i+)$ works not only as a refinement, but also as an assumption. In fact, it implicitly forces Lt_i+ and $Aout_i+$ always to occur in sequence. This assumption is acceptable if the delay inserted between $Rout_i$ (in the simple circuit) or A_i (in the semi-decoupled circuit) and Lt_i is the same for every stage. If this is the case, in the simple circuit, the time interval between $Rout_i+$ and $Aout_i+$ is greater than the time interval between $Rout_i+$ and Lt_i+ . In fact, the former consists of the internal delay of the C-gate in the $(i+1)$ -th stage plus the delay inserted between $Rout_{i+1}$ and Lt_{i+1} whereas the latter consists of just the delay inserted between $Rout_i$ and Lt_i . Analogously, in the semi-decoupled circuit, the time interval between A_i+ and $Aout_i+$ is greater than the time interval between A_i+ and Lt_i+ .

After refining the 3-stage controller as shown in Figure 4, the resultant $TR_{1,2}$ process may be easily analysed to catch performance properties. The $Ps(a, b)$ process given in Figure 8(c) models the property that actions a and b cannot synchronise. This means that a and b can never occur simultaneously. We can compose $Ps(full_1, full_2)$ with the $TR_{1,2}$ process as in Figure 4 and check the equivalence

$$TR_{1,2} * Ps(full_1, full_2) \cong TR_{1,2} \quad (3)$$

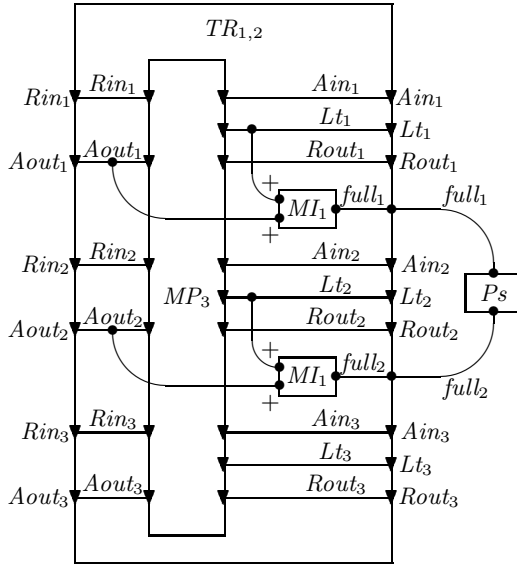


Fig. 9. Analysis of property $Ps(full_1, full_2)$.

This is an instantiation of equivalence (E) in Section 3.1. If equivalence (E) holds then $full_1$ and $full_2$ never occur simultaneously in $TR_{1,2}$. This means that no point in the time interval between Lt_1+ and $Aout_1+$ can occur simultaneously with a point in the time interval between Lt_2+ and $Aout_2+$, that is the two time intervals are disjoint. Therefore, it means that the first two stages can never be full at the same time.

With the Circal System we can automatically verify that using the simple control circuit at most alternate stages can be occupied at the same time. In fact the Ps property holds for adjacent stages, but not for alternate stages. Therefore the degree of parallelism achieved is only 50% of the potential parallelism. This is equivalent to a throughput not greater than 50% [14].

If we check property Ps on a micropipeline of semi-decoupled control circuits, the property does not hold for any pair of stages. This proves that adjacent stages may be occupied at the same time. So no upper bound of 50% is given to the throughput. However, we would like to know how many stages can be occupied at the same time. To achieve this, we need to use the view $V(full)$ in Figure 3.1, in which the i -th stage is refined using the MI_{n-i} process, with n indicating the number of stages in the micropipeline. The generic MI_i process starts marking the time intervals from the i -th occurrence. MI_1 is defined in Figure 3.1(a) and MI_2 is defined in Figure 3.1(b). MI_3 may be defined analogously. In this way we start marking intervals only when the dataflow reaches the output of the micropipeline, that is, when all stages may be occupied.

We want to verify whether this potential full parallelism among all stages can be effectively achieved. Since $full$ belongs to the sort of every instantiation

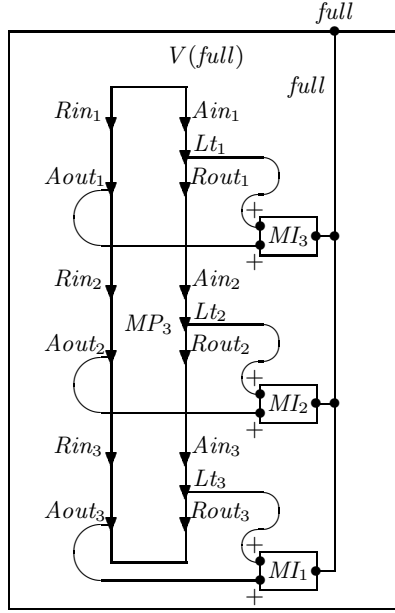


Fig. 10. Abstract view $V(full)$ of a 3-stage controller for the analysis of property $Pp(full)$.

of MI_i , $i = 1, 2, 3$, in Figure 10, according to the composition rules introduced in Section 3 the time intervals that characterise when stages are full have non-empty intersection for some possible execution iff the $full$ action visibly occurs in the behaviour of $V(full)$. In particular, if there is an execution such that all stages are full simultaneously in every handshake cycle, then the $V(full)$ view is equivalent to the $Pp(full)$ process defined in Figure 1(d), which models an infinite sequence of $full$ actions. Using the Circal System we can prove that

$$V(full) \cong Pp(full) \quad (4)$$

and verify that a micropipeline of semi-decoupled latch control circuits has a possible execution such that all stages are full simultaneously in every handshake cycle. Therefore the semi-decoupled latch control circuit shows a possible throughput of 100%. Whether such a performance is effectively attained depends on the environment where the micropipeline operates. In a simple FIFO the maximum performance may be reached, but if the micropipeline incorporates processing logics there may be a performance degradation [4].

4 Discussion

In this paper we have presented a process algebra approach for the integrated verification of correctness and performance in concurrent systems. The verification

procedure is entirely performed within the Circal process algebra, without any recourse to other formalisms, such as temporal logic, stochastic models, dense time models. Both correctness and performance properties are captured in the same verification framework and proved automatically. The automatic checking mechanism is based on both the underlying semantics [14] and a notion of testing equivalence [15]. If the automatic equivalence check fails to match, the Circal System gives a diagnostic in terms of one event trace up to the point where the mismatch has been found [16]. The capability of expressing both model and properties within the same formalism is common in axiomatic frameworks [17], but not in process algebra-based frameworks. Some process algebras do not support this approach due to their use of a binary composition operator [18]. We exploit the multi-way feature of Circal composition in a manner analogous to logical conjunction. Our approach is similar to Roscoe’s approach using CSP [19].

The approach has been applied to two four-phase asynchronous micropipelines. Both micropipelines have been proved to be correct, but they have shown different performance properties. The performance of the simple circuit in Figure 1(a) shows a throughput not greater than 50%. The performance of the semi-decoupled circuit in Figure 1(c) shows a possible throughput of 100%, but such a performance is effectively attained only in the appropriate operational context.

It is interesting to notice that the two performance properties that we have analysed in Section 2.2 can be seen as two temporal properties that belong to two distinct classes. For instance, the property that “adjacent stages can never be occupied at the same time” belongs to the class of safety properties that assert that “for any possible execution something is true at any time”. This class is represented in branching time temporal logic by instantiations of the formula $\forall G\alpha$. In our methodology this class of properties is verified by checking whether or not the process that models the property constrains the system.

A final remark is that our performance analysis is based on a qualitative notion of performance, that is the degree of parallelism of the system components. Such an abstract notion does not provide a complete performance evaluation of the system under analysis. We have seen that the increase of parallelism is associated with an increase in the size and complexity of the handshake control circuit. This might lead to a longer delay in the control circuit and, consequently, to a lower system performance.

Acknowledgments

We would like to thank David Kearney for interesting discussions, Steve Furber for detailed answers to our questions and Graeme Smith for helpful comments on this work. This work has been supported in part by the Australian Research Council, in part by the Information Technology Division of the Australian Defence Science and Technology Organisation and in part by Sun Microsystems Laboratories, USA.

References

1. F. Bacelli et al. *Synchronisation and Linearity — Algebra for Discrete Event Systems*, Wiley, 1992.
2. A. Bailey, G. A. McCaskill and G. J. Milne. An Exercise in the Automatic Verification of Asynchronous Designs. *Formal Methods in System Design*, Vol. 4, No. 3, pp. 213–242, 1994.
3. T. Bolognesi, and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, Vol. 14, No. 1, pp. 25–59, 1987.
4. A. Cerone, D. A. Kearney and G. J. Milne. Integrating the Verification of Timing, Performance and Correctness Properties of Concurrent Systems. In *Proc. of the Int. Conference on Application of Concurrency to System Design*, Aizu-Wakamatsu City, Japan, pp. 109–119, IEEE Comp. Soc. Press, 1998.
5. A. Cerone and G. J. Milne. Modelling a Subclass of CMOS Circuits using a Process Algebra. In *Proc. 6th Annual Australasian Conference on Parallel and Real-Time Systems (PART'99)*, Melbourne, Australia, pp. 386–397, Springer-Verlag, Berlin, 1999.
6. T. A. Chu, C. K. C. Leung and T. S. Wanuga. A Design Methodology for Concurrent VLSI Systems. In *Proc. of ICDD*, pp. 407–410, 1985.
7. R. de Nicola and M. C. B. Hennessy. Testing Equivalence for Processes. *Theoretical Computer Science*, Vol. 34, No. 1/2, pp. 83–134, 1984.
8. S. Donatelli, J. Hillston and M. Ribaud. Comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. In *Proc. of the 6th Int. Work. on Petri Nets and Performance Models*, IEEE Comp. Soc. Press, 1995.
9. S. B. Furber and P. Day. Four-Phase Micropipeline Latch Control Circuit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 4, No. 2, pp. 247–253, 1996.
10. S. B. Furber and J. Lin. Dynamic Logic in Four-Phase Micropipelines. In *Proc. of the 2nd Int. Symp. on Adv. Research in Asynchronous Circuits and Systems*, IEEE Comp. Soc. Press, 1996.
11. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*, Cambridge University Press, 1993.
12. C. A. R. Hoare. *Communication Sequential Processes*, International Series in Computer Science, Prentice Hall, 1985.
13. G. J. Milne. *Formal Specification and Verification of Digital Systems*, McGraw-Hill, 1994.
14. R. Milner. *Communication and Concurrency*, International Series in Computer Science, Prentice Hall, 1989.
15. F. Moller. The Semantics of Circal, Technical Report HDV-3-89, University of Strathclyde, Department of Computer Science, Glasgow, UK, 1989.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*, International Series in Computer Science, Prentice Hall, 1998.
17. I. E. Sutherland. Micropipelines. *Com. of ACM*, Vol. 32, No. 6, pp. 720–738, 1989.
18. web page. Formal Verification of Asynchronous Micropipelines.
<http://www.it.uq.edu.au/~antonio/research/hardware/micropipeline.ntm>.
19. T. Williams. Analyzing and Improving the latency and throughput performance on self-timed pipelines and rings. In *Proc. of the IEEE Int. Symp. on Circuit and Systems*, New York, IEEE Comp. Soc. Press, 1992.

A Methodology for Large-Scale Hardware Verification

Mark D. Aagaard¹, Robert B. Jones², Thomas F. Melham³,
John W. O’Leary², and Carl-Johan H. Seger²

¹ Performance Microprocessor Division, Intel Corporation,
RA2-401, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA

² Strategic CAD Labs, Intel Corporation
JFT-104, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA

³ Department of Computing Science, University of Glasgow
Glasgow, Scotland, G12 8QQ

Abstract. We present a formal verification methodology for datapath-dominated hardware. This provides a systematic but flexible framework within which to organize the activities undertaken in large-scale verification efforts and to structure the associated code and proof-script artifacts. The methodology deploys a combination of model checking and lightweight theorem proving in higher-order logic, tightly integrated within a general-purpose functional programming language that allows the framework to be easily customized and also serves as a specification language. We illustrate the methodology—which has proved highly effective in large-scale industrial trials—with the verification of an IEEE-compliant, extended precision floating-point adder.

1 Introduction

Functional validation is one of the major challenges in chip design today, with simulation and testing a dominating element of the design effort [1]. Throughout the 1990s, formal verification [2] has emerged as a promising complement to simulation. A notable success is equivalence checking using BDDs, now widely-used for checking consistency between adjacent levels in the design flow. Research on the broader problem of functional validation has also delivered promising results in trials on industrial-scale designs [3, 4, 5].

Although algorithmic advances have increased the reach of formal verification, they have still failed to close the gap between the capability offered by verification ‘point-tools’ and modern design complexity. In response, we have coupled our research on verification technology and tools with research into verification *methodology*. The aim is to devise a systematic approach to organizing the activities undertaken in large-scale verification efforts and structuring the associated code and proof-script artifacts.

Algorithmic and tool research primarily address the well-known problem of model-checking *capacity* limits, while often overlooking the equally important

problem of managing the *complexity* of the verification activity itself. Like others, we attack capacity problems with technical innovations in the core verification algorithms. But almost any serious verification effort faces many practical difficulties other than capacity. For example, it almost certainly requires us to break a problem down into many model-checking runs—frequently many hundreds. Organizing all the cases to be considered into a coherent whole or even specifying them clearly (let alone discovering them) is complex, intellectually demanding, and error-prone.

Our methodology addressed this particular verification complexity problem by generating and organizing model-checking runs in a systematic way. More generally, the methodology gives guiding structure and sequence to the many interdependent and complex activities of a large verification effort. The methodology aims, on the one hand, to face the messy realities of design practice (e.g. rapid changes and incomplete specifications) and, on the other hand, to produce high-quality results that are understandable, maintainable—and possibly even reusable. In Section 1 of this paper, we detail aspects of this methodology that are critical for successful application of formal methods to large designs.

Our approach is applied in Forte, a formal verification environment that combines an efficient linear-time logic model checking algorithm (symbolic trajectory evaluation—STE [1]) with lightweight theorem proving in higher-order logic. These are interfaced to and tightly integrated with FL, a general-purpose functional programming language in the ML family [2]. This allows the environment to be customized and large proof efforts organized and scripted effectively. FL also serves as an expressive specification language extending the temporal logic primitives of STE. Section 1 gives a brief overview of Forte and supplies pointers to further information.

The methodology we advocate is separated into the following four distinct but overlapping activities: understanding and encapsulating the circuit, scalar verification, symbolic model-checking, and theorem proving. Each phase has a specific purpose and associated tasks, together with definable *artifacts* that result from these tasks. Section 2 provides a concrete illustration of these phases applied to the verification of an IEEE-compliant, extended precision floating-point adder.

2 Methodology

One of the defining aspects of circuit design is a complex set of trade-offs between design requirements. Different and often contradictory requirements exist—fast circuits, low power, small area, and efficient manufacturing testing. Creating an effective formal verification *methodology* can be as difficult as the process of design. An effective verification methodology must also meet several requirements:

- *Realism*. It cannot depend on resources that are not available in the design environment. For example, complete specifications are usually not available, and access to design engineers may be limited.

- *Incrementality and recoverability.* Preliminary results are needed early in a verification effort. There should be a smooth transition between simulation of special cases and a full proof, so that the effort spent delivers ‘debugging value’ very early on. If changing a specification, circuit, or library causes a previously-passing proof to fail, it should be possible to use incomplete proofs from earlier in the effort to help isolate the problem.
- *Transparency and confidence.* A methodology (and system) should be transparent, i.e. the verification engineer should know what has been proved and what has not. The methodology should also be *sound*, so that false positives are not possible.
- *Structure.* An effective methodology imposes structure on the overall verification effort. This not only helps new users learn, but also increases the productivity of experienced users.
- *Top-down/bottom-up.* A realistic methodology must support a mix of bottom-up and top-down techniques. The subtle features of designs and the capacity limits of model-checking are discovered through bottom-up exploration. Overall problem reduction is achieved by top-down decomposition with case splitting, induction strategies, and algorithm-specific techniques.
- *Debugging and feedback.* The bulk of any verification effort is debugging, so it is crucial to optimize the verification environment for proof *failure*, not success. Not only must the system quickly discover failures, it should provide focused feedback that enables a tight, rapid debug loop.
- *Regression.* The methodology should produce verification artifacts that are easy to maintain and adapt to changing specifications and designs. Test cases from initial proof development should continue to be usable in exploring these changes.
- *Effort reuse.* Verification is an expensive, human-intensive activity. Proof reuse should be supported to amortize the verification cost over design changes and even multiple design efforts. Specifications and high-level decomposition strategies are particularly important candidates for reuse.

The methodology detailed in this paper strikes a balance between these different requirements. It is divided into four distinct but overlapping phases of effort, which we introduce in the remainder of this section.

The first phase of the methodology is circuit *wiggling*—the process of educating oneself and one’s tools about the circuit and its operating environment. The process begins by identifying an initial set of important signals. This is done by using a simulator with a ‘don’t care’ value (called ‘X’) to observe the effect of driving the circuit’s input and state signals. We say a circuit is ‘wiggling’ when enough knowledge has been gained about how to drive these signals to make defined (non-X) values appear at the circuit outputs. Efforts during this phase are not concerned that the outputs are correct, just that they appear as non-X values. But understanding the circuit’s functionality at even this crudely abstract level evolves into an initial sketch of the specification.

Aside from knowledge, the primary artifact of this phase is a circuit API that defines an interface to the circuit nets and their timings. Concretely, the API is a functional program (written in FL) that encapsulates the details of the circuit's timing and I/O protocol, allowing the functional specification and other later developments to abstract away from these details.

Targeted scalar verification, the second phase, begins by focusing on scalar (i.e. bit-pattern) input and state values selected to invoke easy-to-understand circuit behavior. The primary difference between wiggling and verification is that verification entails checking the circuit against a specification. Specifications are constructed incrementally. Simple portions are written first, using obvious scalar input and state vectors to check agreement with the circuit. As more of the required functionality is understood, and as the simple parts of the specification are debugged, additional functionality is added. As the specification is filled out, the scalar verification cases used are chosen to target several representative samples from each region of the input and state space explored, including boundary conditions (e.g. maximum and minimum values, zeros).

The main artifacts of this phase are a functional specification, an improved circuit API, and a set of scalar test vectors (which are saved to support regression testing). Discrepancies between the circuit and specification are usually a result of errors in the circuit API or the specification itself. But even at this early stage genuine design bugs could be discovered. When it becomes time consuming or difficult to find stimuli that result in discrepancies, it is time to move to symbolic model checking.

Symbolic model checking is the third phase of verification and marks the beginning of serious formal verification efforts. Instead of driving circuit nets with scalar values, we present the circuit with symbolic values represented by BDD variables and compute a symbolic representation of the resulting outputs. The model-checking engine being employed automatically verifies the resulting input/output relation against the specification, and in case of disagreement it generates scalar counterexamples—or even complete symbolic characterizations of the difference. (In fact, the algorithms in Forte are more subtle than this and operate incrementally.) Symbolic input values can be restricted by constraints, which are described by FL specifications developed in the course of the proof. For model-checking efficiency, these are encoded into parametric input functions.

Most hardware bugs are found during this phase of verification. Once symbolic model checking begins in earnest, BDD sizes will likely confront the capacity limits of the tool. This phase therefore includes the significant challenge of managing BDD complexity. It is often necessary to find good variable orderings, which is accomplished by a combination of automatic and manual techniques. This phase of effort also involves determining the limits of what can be checked by the model checking engines.

¹ We borrow the term API, *Application Program Interface*, from software engineering.

² Our use of 'symbolic' here differs from its meaning in SMV-style *Symbolic Model Checking*.

The main artifacts produced in this phase are an improved specification, BDD variable orderings, and a set of constraints that characterize regions of the input space for which model-checking is feasible. All these are realized as FL programs, and are easily inspected and modified by the user.

When capacity management with variable ordering provides diminishing returns, a verification decomposition strategy must be developed. This is where the worlds of model checking and theorem proving meet.

The final, *theorem proving* phase is to mechanically check the correctness of the specification and the soundness of the decomposition strategy. In addition to organizing the model checking decomposition, theorem proving helps the verification engineer discover missing details in the specification. This phase does not directly find bugs in the circuit. Instead, bugs in the proof may be detected, such as missing cases or incorrect reasoning. Fixing these bugs may require modifying the model-checking cases, which may then reveal circuit bugs.

In addition, the theorem proving phase may include proof-based analysis of the specification, typically by using a theorem prover to derive high-level properties from the specification. Because it will have been created in a bottom-up manner, there is a danger that the specification simply reverse engineers the circuit—including its bugs. The aim of this activity is therefore to gain extra confidence in the correctness of the specification by assessing it against properties independent of the circuit. These could be derived, for example, from standards such as the IEEE floating-point specification, the PCI bus specification, or a programmer's reference model for a microprocessor.

The artifacts of this phase are the final version of the functional specification, a top-level correctness statement, a collection of model checking runs, and a mechanized proof connecting the top-level correctness statement to the model checking runs. There may also be a collection of proofs of properties derived from the specification.

The four phases of verification are not strictly sequential—a fair amount of overlap and backtracking happens in practice. This is shown in Figure 1, a schematic time-line of how a typical verification might proceed through our methodology. Reading from top to bottom, the diagram shows the relative distribution of time spent on the major activities as a typical verification effort moves through the four phases of the methodology. The curves shown are, of course, only a rough guide; the exact distribution of effort will depend on the specific verification project. Along the right-hand side of the diagram, definite criteria are listed for transitions between phases. The diagram also shows the point at which the main artifacts are finalized. Section 1 will illustrate each phase in the context of our motivating example.

Finally, finishing the theorem proving phase by no means terminates involvement with the artifacts of the proof effort. If the design is still 'live', any part of the verification code (from API on upwards) may have to be adapted to track design changes. The structure imposed on these artifacts by our methodology helps to localize changes textually. After design changes, scalar and symbolic simulations, model checking runs, and theorem proving scripts all need to be

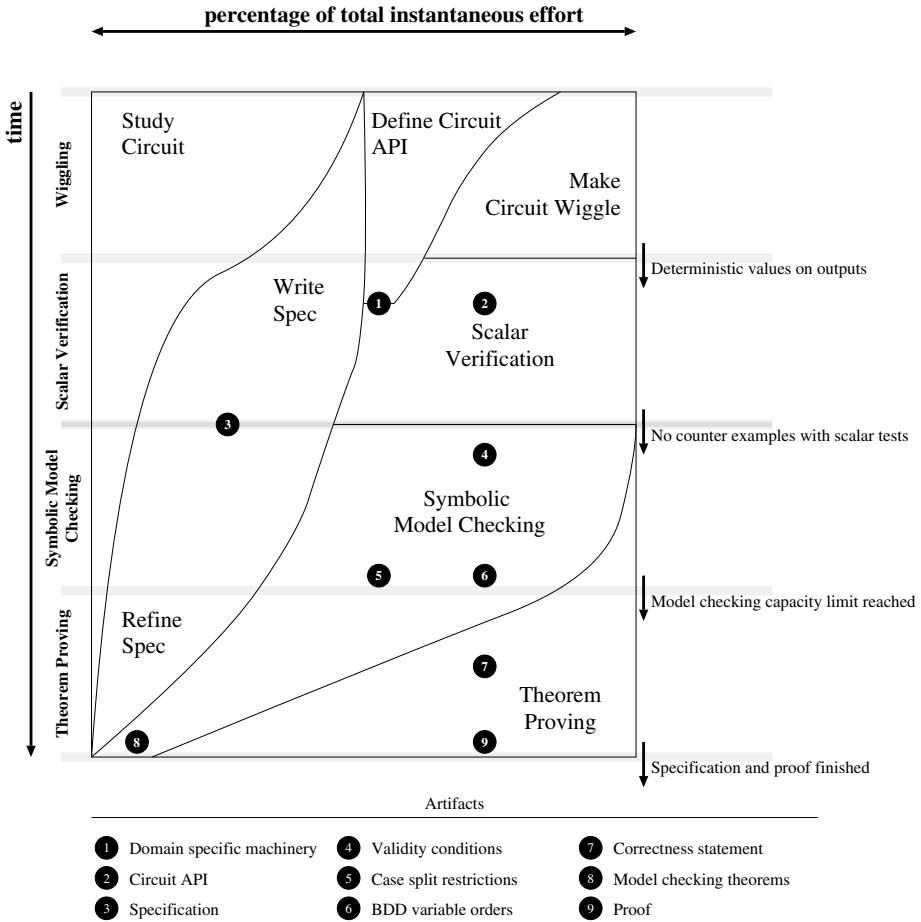


Fig. 1. Schematic time-line of verification activities.

re-run. In Forte, the concrete form of FL programs provides maintainable and incrementally executable scripts for all these verification activities. The results of one proof effort may also be re-used on future designs. If the algorithm does not change dramatically, this can be as easy as replacing the API.

3 The Forte Verification Environment

An effective methodology is obviously dependent on an effective verification platform. The methodology ideas described in this paper are embodied in Forte, our custom-built verification environment. Forte integrates model checking engines, BDDs, circuit manipulation functions, theorem proving, and a functional programming language. It builds formal circuit models from standard HDL sources,

and includes tightly integrated graphical tools for the display of circuit structures and waveforms.

In other publications, we have presented details about Forte’s approach to combining model checking and theorem proving [10], symbolic simulation [11], and arithmetic verification [12]. The present paper focuses on our verification methodology, so we provide only a brief overview of Forte here. The key capabilities of Forte for this paper are the FL language, symbolic trajectory evaluation, and theorem proving.

The FL Language

FL is a strongly-typed, lazy, functional programming language. Syntactically, it borrows heavily from Edinburgh-ML [13]. Semantically, its core is similar to lazy-ML [14]. One distinguishing feature of FL is that BDDs are built into the language and every object of the boolean type `bool` is represented as a BDD.

The FL language lies at the very center of Forte. Through its embedded BDD package and primitive or defined functional entry-points, it provides a flexible interface for invoking and orchestrating model checking runs. It is also used as an extensible ‘macro language’ for expressing specifications, which are therefore human-readable but when executed compute efficiently checkable properties in a low-level temporal logic. Finally, it provides the control language for Forte’s theorem prover and—through the concept of *lifted FL* [15]—the primitive syntax of its higher order logic.

This approach gives a generic, open framework where solutions can be tailored to individual verification problems. For each verification effort or project, the user ‘programs up’ in the FL functional language just the right specification constructs for the problem domain, as well as scripts for orchestrating verifications. By structuring these according to our methodology, much of this work can be reused over entire classes of verifications.

Symbolic Trajectory Evaluation

Symbolic trajectory evaluation (STE) is a formal verification method that can be viewed as a hybrid between a symbolic model checker and a symbolic simulator [16]. As a simulator, it can compute the result of executing a circuit with concrete boolean test vectors as inputs; as a *symbolic* simulator, it can compute symbolic expressions giving outputs as a function of arbitrary inputs; as a model-checker, it can automatically check the validity of a simple temporal logic formula for arbitrary inputs—computing an exact characterization of the region of disagreement in case the formula is not unconditionally satisfied. STE’s seamless connection between simulation and verification is crucial to satisfying our methodology’s requirement for early results.

STE performs model checking with a symbolic simulation-based algorithm that is significantly more efficient than more traditional symbolic model-checking approaches. STE is particularly well suited to handle datapath properties, and

it is used in our work to verify gate-level models against more abstract reference models. However, since symbolic trajectory evaluation is based on BDDs, there are circuit structures that cannot be handled monolithically. For example, multipliers are beyond the capability of STE alone.

Theorem Proving

Because model checkers are incapable of verifying certain circuit structures, and in any case have inescapable capacity limits, most verifications must be broken up into pieces. A decomposition typically either follows the structure of the circuit or partitions the input data space (though other problem-reduction strategies are also possible). To ensure that no mistakes are made in this partitioning process and to check that no verification conditions are forgotten, a mechanically-checked proof is needed.

In Forte, a lightweight theorem proving system called ThmTac is used for this task. It is implemented in FL and tightly integrated with the Forte model checkers; as a result, no translation or re-formulation of the verification results is needed before theorem proving can be performed. ThmTac is loosely based on the classic model of LCF proof systems [14]. It consists of an FL implementation of a Church-style formulation of higher-order logic, with an extensible theorem-proving infrastructure built on top.

Though the mechanism of lifted FL, the logical language of Forte actually shares an implementation of the underlying λ -calculus data structures with FL programs themselves. This means that lifted FL also allows the FL evaluation machinery to be invoked as an inference of the logic; essentially, any phrase that evaluates to true in FL can be ‘lifted’ to be a theorem in the logical language. In particular, a successful STE (or CTL) model-checking run in Forte is just a specific kind of function call that evaluates to true in FL. Any such model-checking run can therefore be lifted to a higher-order logic theorem asserting its logical content, providing our methodology with a very smooth and flexible link between model checking and theorem proving.

On top of this language infrastructure, ThmTac has numerous FL libraries implementing our own versions of the conventional theorem-proving technology—tactics and tacticals, term rewriting, and so on. The ThmTac theorem-prover also has several special-purpose, integrated decision procedures, making reasoning (particularly about arithmetic results) significantly easier and more efficient. For ease of use and proof robustness, it also supports the ‘declarative’ proof-construction style [14].

4 Verifying a Floating Point Adder

In this section we will illustrate the application of our methodology in the verification of a floating-point adder. The adder performs IEEE-compliant floating-point addition and subtraction at single, double, and extended precisions, and

supports four rounding modes—toward 0, toward $-\infty$, toward $+\infty$, and to nearest. It was verified as part of a large-scale verification effort undertaken on the Intel Pentium® Pro processor [4].

Sections 4.1 to 4.4 show how the adder verification proceeded through the four phases of our methodology—understanding the circuit and making it ‘wiggle’, targeted scalar verification, symbolic model checking, and theorem proving.

4.1 Wiggling

The verification effort began with education about the circuit and its operating environment. This was accomplished by reading design documentation, perusing the RTL code, consulting other verification and validation engineers, and consulting the circuit designers. Notes from this activity evolved into an initial sketch of the circuit API, specifying the names and timing of circuit signals that are important to the verification.

Once an initial set of important signals has been identified, the process of getting the circuit up and *wiggling* begins, based on the use of symbolic trajectory evaluation as a scalar simulator. The aim of this is to develop an FL circuit API that acts as a kind of simulation ‘test harness’ for driving input signals and observing output signals. Developing this artifact involves discovering a mass of detail about the input and output protocols, including the exact timing of important signals, and encapsulating this in cleanly-structured FL code.

For the floating-point adder, the starting point was a small set of simulations on very simple input cases (for example, computing zero plus zero). The aim was to derive information about the relationship between the control inputs, state nodes, and outputs of interest to supplement what was available in the design documentation. Wiggling is a highly interactive process, with information being gleaned by analyzing circuit behavior with a waveform viewer, viewing circuit structure with a graphical browser, and writing FL functions and predicates that probe the circuit.

After it was understood how to drive the input and state signals in such a way as to reliably cause defined (non-X) values to appear at the outputs, we refined the API in order to reduce the number of signals that are driven and the length of time that they were driven. The aim was to make fewer assumptions about the input behavior, in order to strengthen the eventual verification result. Like the initial phase of wiggling, identifying the weakest necessary assumptions is an iterative process that benefits from further discussion with designers and architects, and study of the behavior of the circuit.

The final circuit API for the adder consists of two FL functions, one for inputs and one for outputs. The first function, `fadd.fsub.protocol`, describes the behavior of the circuit’s inputs during an addition or subtraction operation. It takes the following arguments: `uop`, the bit-vector opcode of the instruction to be executed (FADD or FSUB in this example); `pc` and `rc`, the precision and rounding mode of the operation; and `A` and `B`, the two floating-point operands.

```

let fadd_fsub_protocol [uop, pc, rc, A, B] =
  // Drive clocks and resets
  gen_clocks and no_resets and
  // Opcode and inputs
  (( (opcod isv uop) and      // Opcode
     (opcodv is T) and       // Opcode is valid
     (s1 isv A) and          // Operand 1
     (s1v is T) and          // Operand 1 is valid
     (s2 isv B) and          // Operand 2
     (s2v is T))             // Operand 2 is valid
    in_cycle 2) and
  // Round and precision control follow inputs by one cycle
  (( (roundc isv rc) and      // Rounding mode
     (precc isv pc))          // Precision control
    in_cycle 3);

```

Within this API, we impose the conditions that the circuit is clocked correctly and that there is no reset during the execution of an operation. The function also drives the circuit’s input nets with the opcode, the two input operands, and the rounding and precision controls—each in the correct clock cycle. Inside the functions called in the above definition is a mapping between signal names in FL and the actual node names in the source RTL, as well as a layer of code that establishes a unit-delay temporal abstraction.

The second API function, `fadd_fsub_result`, just observes the output `wbdata` at the appropriate clock cycle. It also asserts that the valid bit `wbdatav` is set.

```

let fadd_fsub_result [res] =
  (( (wbdata isv res) and
     (wbdatav is T))
    in_cycle 5);

```

Once the API has been defined and validated, it establishes an abstract view of the circuit’s I/O interface upon which the rest of the verification can build. This clean structuring mechanism helps makes higher levels of the proof effort reusable. And because the various mappings in the API are realized as FL source code, they can easily be inspected and understood if necessary—supporting the transparency requirement of our methodology.

4.2 Targeted Verification

Targeted verification begins essentially with regression testing, using the simple input cases from the wiggling phase. The difference between wiggling and verification is that verification entails checking the circuit against a specification. The objectives of this stage are a refined circuit API and a specification of the circuit functionality.

The specification to be developed is incorporated into the output part of the API in the previous section. For the adder, the function `fadd_fsub_result`

was revised to take as arguments `fspec`, a specification of the function to be performed by the datapath, and the same five input values as the input API.

```
let fadd_fsub_result fspec [uop, pc, rc, A, B] =
  // fspec maps inputs -> result
  let res = fspec [uop, pc, rc, A, B] in
  (wbdata isv res) and
  (wbdatav is T)
  in_cycle 5;
```

Based on the output values supplied, the revised API function uses `fspec` to compute the expected result `res`, and asserts that whatever is observed on the circuit's output signals must be identical to this.

Note that `fadd_fsub_result` is a higher-order function; its argument `fspec` is itself a function. The revised API makes no assumptions about the particular computation done by the specification; it could therefore be reused as the API for operations other than addition and subtraction. The use of a full-fledged programming language is what enables clean abstractions like the circuit API to be constructed.

Our specification of floating-point addition and subtraction was a straightforward adaptation of a textbook algorithm [10]. Initially, our specification supported double precision operations and rounding toward zero only, and was tested only for ‘true’ addition (that is, addition of operands with like signs, or subtraction of operands with different signs). Through verification aimed at particular corner cases, our specification was extended to support single, double and double-extended precisions, and four rounding modes.

Our final specification was the composition of five functional stages, shown schematically in Figure 2. In the first stage, the operands are inspected and the mantissa of the smaller operand is shifted right in preparation for addition. Addition or subtraction of the mantissas takes place in the second stage. Following addition, the mantissa is normalized—shifted left or right to align its significant bits with the binary point. Finally, the mantissa is rounded and truncated according to the given rounding mode and precision and renormalized if required.

We will not show the complete specification here, but the rounding function `RND` is illustrative of how a specification evolves in this phase of the methodology. The FL function `RND` takes as arguments the sign bit `s` and significand (or ‘mantissa’) `sgf` of the number to be rounded, as well as the precision and rounding controls:

```
let RND pc rc s sgf =
  // Extract lsb, guard, round sticky bits.
  let L = Lsb pc sgf in
  let G = Guard pc sgf in
  let RS = RoundS pc sgf in
  // Conditionally add one to LSB
  let rbit =
```

```

        (rc '=' TO_ZERO)      => F
    | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
    | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
    | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
    | F in
// Result truncates mantissa to precision specified
// by pc, adds rbit and pads result with zeros.
Result rbit pc sgf;

```

In our initial specification, RND simply truncated its result to the required number of bits (rounding toward zero). Later, as the other rounding modes were added, the concepts of guard and sticky bits were introduced and used. Each extension was justified by a particular set of test inputs.

For example, $1.0 \times 2^{30} - 1.0 \times 2^0$, in single precision, should yield 1.0×2^{30} when rounded toward $+\infty$ and $1.11111111111111111111111111111111 \times 2^{29}$ when rounded toward $-\infty$. The correct behavior for such cases was usually deduced using pencil and paper, and was always checked against the actual behavior of the circuit. As a guard against the danger of inadvertently incorporating circuit bugs in our specification, we will later verify the specification itself against the behavior required by the IEEE 754 standard. Section 4.4 describes this aspect of the verification.

Because our specification is written in FL, it is fully executable. It also contains no temporal information, mentions no signal names from the RTL description, and says nothing about interface protocols. The circuit API supplies this information for a given implementation of the floating-point addition/subtraction algorithm. Decoupling the specification of the datapath functionality (which can remain constant for the life of a design) from the names of signals and their timing (which can change from week to week in a design project) supports our requirements of regression and reuse. Again, the use of FL is a key enabler.

Work in this phase is directed towards defining and debugging a specification, and so involves more thinking about the actual computation than in the wiggling phase. It still consists mostly of simulation, but more time is spent analyzing simulation results from the circuit and specification and less examining the internals of the circuit. When it becomes time consuming or difficult to find stimuli that result in a discrepancy between the circuit and specification, it is time to move on to symbolic model checking.

4.3 Symbolic Model Checking

The symbolic model checking phase marks the beginning of serious formal verification efforts. At this point symbolic Boolean (BDD) values are declared for each circuit input:

```

let OPCOD = variable_vector "opcod[opcod_w:0]";
let PC    = variable_vector "pc[1:0]";
let RC    = variable_vector "rc[1:0]";

```

```

let A      = variable_vector "A[fp_w:0]";
let B      = variable_vector "B[fp_w:0]";

```

Our eventual aim in this phase is to drive all input signals with symbolic Boolean values, and thus obtain exhaustive confirmation that the circuit conforms to its specification. However, STE also allows an arbitrary mix of scalar and symbolic simulation values, and so satisfies the methodology requirement for incrementality.

This phase brings with it two major challenges, the first of which is the capture of input validity conditions. Many circuits impose constraints on their environments and are guaranteed to work correctly only if those conditions are met. For example, circuits associated with decoding instructions may require that their inputs are legal instructions, or a non-pipelined execution unit may require that there is a certain latency between consecutive operations.

The floating-point adder also imposes constraints on its environment; each of its operands must either be zero (all exponent and mantissa bits zero) or normal (with exponent between 0 and maximum, and the mantissa of the form $1.b_1b_2b_3\dots$). The FL function `VALID_FP` expresses this constraint:

```

let isNORM fp  = (exp fp '> 0) AND (exp fp '< MAX_EXP) AND
                  (Jbit fp);
let isZERO fp  = (exp fp '= 0) AND (man fp '= 0);

let VALID_FP fp = (isZERO fp) OR (isNORM fp);

```

In addition to data validity conditions, a huge variety of other conditions will typically be used to restrict the input and state spaces for a verification. These include isolation of the case that is being executed, cases that are believed to be buggy, and cases that are not yet included in the specification.

For the floating-point adder verification, we are interested only in the response of the circuit to `FADD` and `FSUB` opcodes. The validity condition therefore restricts the opcode, as well as ensures each operand is either zero or normal:

```

let Add = (OPCOD = FADD);
let Sub = (OPCOD = FSUB);

let ValidInput = (Add OR Sub) AND VALID_FP A AND VALID_FP B;

```

FL allows a user to implement whatever specialized ‘vocabulary’ of functions is needed to express input constraints like these in the most natural way. Here we have used FL to define a concise vocabulary for the domain of floating-point verification. This supports our goal of transparency, as definitions like `VALID_FP` and `ValidInput` can easily be inspected and understood. Definitions like these also provide infrastructure that all users can employ in later verification efforts.

The second major challenge in this phase is complexity management. Once symbolic model checking begins in earnest, BDD sizes will begin to exceed the capacity limits of the tool. Some reduction in BDD sizes can be obtained by

choosing good variable orderings, but a top-level strategy to combat verification complexity becomes crucial. For the floating-point adder, we divided the verification into numerous subcases—initially according to the whether the mantissa datapath performs addition or subtraction, and then according to the difference between the two exponents. The following definitions characterize the domain of operation of the mantissa datapath as ‘true addition’ or ‘true subtraction’:

```
let EqualSigns = ((sgn A) = (sgn B));

let TrueAdd =
  ValidInput AND
  ((Add AND EqualSigns) OR (Sub AND NOT EqualSigns));

let TrueSub =
  ValidInput AND
  ((Add AND NOT EqualSigns) OR (Sub AND EqualSigns));
```

In each domain of operation, we further decompose the model-checking problem according to the exponent difference. As an example, for ‘true addition’ we consider the following cases:

1. The exponent of operand B is much larger than the exponent of operand A (that is, the magnitude of operand A is almost negligible).
2. The exponent of operand B is larger than the exponent of operand A by 1, 2, ..., n ; where n is the mantissa width.
3. The exponents of the operands are equal (only one case).
4. The exponent of operand A is larger than the exponent of operand B by 1, 2, ..., n ; where n is the mantissa width.
5. The exponent of operand A is much larger than the exponent of operand B.

We compute the cases systematically in FL, as well as describe them in a transparent form, by defining auxiliary functions capturing relations between operand exponents. For example, `Exp1BiggerBy` expresses the condition that the exponent of operand A is larger by n than the exponent of operand B. `Exp1TooBig` captures the condition that the magnitude of operand A is almost negligible.

```
let Exp1BiggerBy n = ((exp A '-' exp B) '=' (nat_to_bv n));

let Exp1TooBig =
  ((exp A '>' exp B) AND // No wraparound
   ((exp A '-' exp B) '>' (nat_to_bv max))); // Vin1 >> Vin2
```

Each true addition case is then generated by an FL function `true_add_case` using these functions. Every case imposes the restriction `TrueAdd`, together with a condition generated from an integer that quantifies the exponent difference (and is also used later to compute an appropriate BDD variable ordering.) The function `ExpDiff` constructs an individual case from this integer. Finally, the list `true_add_cases` enumerates all the true addition cases.

```

let ExpDiff n =
  (n < ~max) => Exp2TooBig |
  (n >= ~max AND n < 0) => Exp2BiggerBy (~n) |
  (n = 0) => EqualExps |
  (n <= max AND n > 0) => Exp1BiggerBy n |
  Exp1TooBig;

let true_add_case n = (TrueAdd AND ExpDiff n, n);

let true_add_cases = map true_add_case (~max-1 upto max+1);

```

The FL source code for the case decomposition is a major artifact produced in this phase. For the floating-point adder, the case analysis given above follows an input space decomposition strategy; it is therefore specific to the algorithm, but is *not* dependent on fine details of circuit structure (which are hidden by the API). This artifact is therefore highly reusable in other settings that employ roughly the same algorithm.

Throughout this phase, model-checking is invoked through an FL verification function built on top of the circuit API. The function `prove` verifies one case, and is supplied with an input validity condition and an ordering parameter (the difference between the exponents of the operands).

```

let prove (vc, n) =
  // Drive the inputs
  let ante = fadd_fsub_protocol [uop, pc, rc, in1, in2] in
  // Specification's idea of the output
  let cons = fadd_fsub_result fadd_fsub [uop,pc,rc,in1,in2] in
  // Install ordering, then run symbolic trajectory evaluation
  (Order n) fseq (STE_vc ckt vc ante cons);

```

This is where the circuit API interacts with the specification; the API function `fadd_fsub_protocol` computes the antecedent (stimulus) for a symbolic trajectory evaluation run, and the API function `fadd_fsub_result` is called with the datapath specification `fadd_fsub` to compute the consequent (expected result). `STE_vc` performs symbolic trajectory evaluation upon the circuit `ckt`, incorporating the supplied condition `vc` and the computed antecedent `ante` and consequent `cons`. The function `Order` generates and installs a BDD variable ordering, based on the parameter `n` (the difference between the exponents of the two operands being added or subtracted). The ordering is chosen to mimic the alignment of mantissa bits that will take place in computing the sum or difference of the operands.

Verification of all the true addition cases is accomplished by applying `prove` to the list of all cases:

```

let true_add_result = map prove true_add_cases;

```

`true_add_result` is a list of boolean values, each signifying success or failure of a particular case. In practice, since the cases are independent, we verify them in parallel on a network of workstations.

The case breakdown for true subtraction is similar, with the complication that additional case splits are needed when the exponents are equal or differ by 1. In these cases, the difference between the two mantissas may have many leading zeros and a large left shift may be needed to renormalize. The true subtraction cases are generated with the help of further auxiliary functions in FL.

As verification proceeds in this phase, disagreements between the circuit and the specification are increasingly likely to indicate bugs in the circuit rather than the specification. Of course, subtle bugs may still lurk in the specification. For example, in model-checking the true subtraction cases we discovered a discrepancy between the specification and the circuit when the exponents differed by one and the difference between aligned mantissas was the smallest non-zero value. In this corner case, our specification incorrectly normalized the result before rounding; we corrected this by adding one extra bit to its internal mantissa.

4.4 Theorem Proving

The final, theorem proving, phase of the methodology is relatively open-ended and can include many possible activities. The most common activity is checking the validity of the problem decomposition devised in the model checking phase and ensuring completeness of coverage. Of course, a certain amount of simple checking can be done directly in FL. For example, we can compute in FL a list of all cases verified in the floating-point adder example:

```
let cases = true_sub_cases @ true_add_cases;
```

Then to check that the list is exhaustive, we just compute in FL a boolean (BDD) value `cases_exhaustive` as follows:

```
let all_cases_disjunction = OR_list (map fst cases);

let cases_exhaustive =
  (ValidInput AND (Add OR Sub)) ==> all_cases_disjunction;
```

The logical implication calculated here states that, under the assumptions the inputs are either normal or zero and the operation is legal, the disjunction of the list of all cases comes out true.

We use such mechanized checks because, with complicated specifications and environmental constraints, even painfully detailed code reviews will often miss important errors. In one verification script, we discovered quite late that we had duplicated an opcode in the instruction map, and so were not verifying one type of instruction. In another verification, a simple typo in an environmental constraint caused us to overlook an entire class of instructions.

The theorem proving phase goes well beyond simple computation of checks in FL. It provides a high degree of confidence in the validity and coverage of a given

decomposition, and it can handle decomposition strategies not easily analyzed by just computing BDDs. Moreover, through a computational reflection technique called lifted FL [17], it can also trivially employ the kind of evaluation methods described above as a proof strategy.

The aim of theorem-proving is to use deductive proof to knit together the (possibly hundreds of) individual model checking runs generated in the previous phase into a top-level correctness statement that says that the circuit satisfies the specification for all valid input and states. In the style encouraged by our methodology, a top-level correctness statement concisely ties together the three major elements of the whole proof effort—the circuit, the validity condition on the environment, and the specification. Writing the correctness statement in a standard, stylized form helps satisfy the *transparency* requirement for our methodology. A standard form makes it easier for a reader to extract pieces of critical information and answer questions about the verification. For example, one might ask, ‘*Was this circuit verified for all addressing modes?*’.

Proving a top-level correctness statement does not directly find bugs in the circuit. Instead, bugs may be found in the verification—missing cases, overly tight environmental constraints, or incorrect reasoning steps. Fixing these may require changing the model-checking cases and re-running some or all of them. Of course, these new model-checking runs may then reveal bugs in the circuit.

Technically, theorem proving is seamlessly integrated with model checking in Forte using the mechanism of lifted FL, which allows any FL phrase that evaluates to true to be converted into a theorem of higher order logic. The collection of successful verifications from the model-checking phase all evaluate to true, and so we can use this mechanism to convert their FL sources into theorems. By standard (and rather trivial) reasoning in higher order logic, these theorems can then be combined into the top-level correctness statement. A side-effect will be to have checked the decomposition soundness and case coverage.

The primary artifact of this activity—arguably of the entire effort—is a top-level correctness statement. Theorem proving glues the results of multiple model checking runs together to derive this theorem, which is itself beyond the capacity of model checking. But the final phase of the methodology can also provide an independent check on the ‘quality’ of the specification, by deriving independent properties from it.

For example, the floating point adder specification (which is essentially an algorithm) was validated against the more abstract IEEE standard specification of floating point addition. For this proof, we divide the algorithm into two stages (Figure 1). Stage 1 includes alignment, addition/subtraction, and normalization; stage 2 is the rounder. For each stage we have assumptions about the inputs of the algorithm, properties to prove about the output, and some auxiliary properties to prove that are relied upon by subsequent stages.

The overall properties of Stage 1 are established by ‘cascading’ properties of the alignment, add/subtract, and normalization algorithms. Proofs of this first stage verify p ’s relation to the inputs and, roughly speaking, conclude that

$$(|p| \leq |in2+in1|) \wedge (|in2+in1| < |p|+ulp) \wedge (p_s \equiv (|p| < |in2+in1|))$$

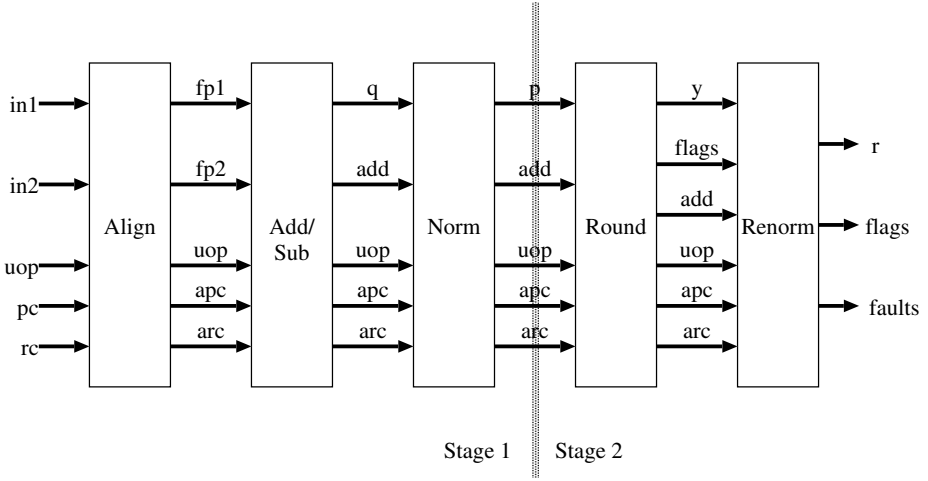


Fig. 2. Decomposition of reference model proof.

The value ulp is the distance between the rational number $|p|$ and the rational of next largest magnitude actually representable with the precision available.

The proof of stage 2 is decomposed into four cases, one for each rounding mode in the IEEE standard. The proof proceeds by assuming the above property of stage 1 and concluding an output specification for each rounding mode. For example, when rounding toward $-\infty$ we have

$$(r \leq \text{in2} + \text{in1}) \wedge (\text{in2} + \text{in1} < r + \text{ulp}) \wedge \\ (s = -1 \wedge m = 2^p \supset r + \frac{1}{2}\text{ulp} > \text{in2} + \text{in1})$$

This says the true sum of the inputs lies between the computed result r and the next representable rational number, as required by the IEEE standard. The third conjunct deals with an important special case. When r is negative ($s = -1$) and r 's mantissa m is a power of two, then the pair of representable values on either side of r will have different exponents. The distance from r to the next largest representable value is then one-half the distance between r and the representable value of *next largest magnitude*.

The proof concluded by combining the theorem for stage 1 and the separate cases for stage 2 to derive one theorem specifying the outputs of the adder in terms of its inputs for all rounding modes (and, indeed, for several precisions).

In this example, the theorem proving phase produces two major results. First, it ensures that the decomposition from the top-level correctness statement to the model checking runs is correct. In principle, this could have been verified by an ‘infinite capacity’ model checker; the second result goes much further. By checking the specification against an abstract IEEE specification, theorem proving is used to derive more a obviously correct theorem—one that is beyond the expressive power of the specification language of the STE model-checker.

5 Conclusions

We have described a methodology for carrying out datapath verification on large hardware systems. The FADD example in this paper uses STE-based model checking; our framework also includes CTL-based model checking. Our verification methodology has evolved over a series of case studies carried out at Intel. These include verifications of an IA32 instruction-length decoder [11, 12] and of IEEE compliance of many of the major Intel Pentium® Pro floating point instructions [13]. The methodology is certainly not complete—industrial hardware verification is very challenging, and much further work remains in methodology and the underlying technology.

Our methodology relies heavily on the capabilities provided by the Forte system. While developing Forte, we have been conscious of the competing goals of capability and usability for the tools. We are also keenly aware that a routine verification for the technology or tool developer may be virtually impossible for others to duplicate. Our methodology aims to address these issues by targeting the usability of Forte and by providing information that will help others to use Forte successfully.

A typical introduction to Forte begins with the new user exploring conventional simulation in Forte. Once the user is comfortable with this mode of usage, symbolic simulation and model-checking are introduced. When the capacity limits of model-checking are reached, the user then learns about theorem-proving. This incremental and modular approach also has benefits in the desired usage model for Forte when verification engineers are using it on a regular basis. Given that there will be much more model-checking than theorem-proving, a validation team composed of a few theorem-proving experts and a larger number of model-checking experts can divide up the verification tasks.

Our methodology aims to create proof efforts that are relatively circuit-independent. This makes our proofs more robust during design evolution, as well as more reusable for future generations of designs implementing the same functionality.

Acknowledgments

Ching-Tsun Chou, Limor Fix, Brian Moore, and Eli Singerman made helpful comments on a draft of this paper. Thanks are also due to the anonymous referees for their comments.

References

- [1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers, 2000.
- [2] T. Kropf, *Introduction to Formal Hardware Verification*, Springer-Verlag, 1999.
- [3] Á. Th. Eiríksson, “The formal design of 1M-gate ASICs,” in *Formal Methods in Computer-Aided Design*, G. Gopalakrishnan and P. Windley, Eds. 1998, vol. 1522 of *Lecture Notes in Computer Science*, pp. 49–63, Springer-Verlag.

- [4] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technical Journal*, First quarter, 1999, Available at developer.intel.com/technology/itj/.
- [5] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall, "Practical application of formal verification techniques on a frame mux/demux chip from Nortel Semiconductors," in *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf, Eds. 1999, vol. 1703 of *Lecture Notes in Computer Science*, pp. 110–124, Springer–Verlag.
- [6] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.
- [7] L. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1996.
- [8] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
- [9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.
- [10] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of boolean constraints," in *ACM/IEEE Design Automation Conference*, 1999.
- [11] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *ACM/IEEE Design Automation Conference*, 1998, pp. 538–541.
- [12] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.
- [13] L. Augustson, "A compiler for lazy-ml," in *ACM Symposium on Functional Programming*, 1984, pp. 218–227.
- [14] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Lifted-fl: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds. 1999, vol. 1690 of *Lecture Notes in Computer Science*, pp. 323–340, Springer–Verlag.
- [15] D. Syme, "Three tactic theorem proving," in *Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds. 1999, vol. 1690 of *Lecture Notes in Computer Science*, pp. 203–220, Springer–Verlag.
- [16] J. Feldman and C. Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*, McGraw-Hill, 1994.

Model Checking Synchronous Timing Diagrams

Nina Amla¹, E. Allen Emerson¹, Robert P. Kurshan², and Kedar S. Namjoshi²

¹ Department of Computer Sciences, University of Texas at Austin***
{namla,emerson}@cs.utexas.edu

<http://www.cs.utexas.edu/users/{namla,emerson}>

² Bell Laboratories, Lucent Technologies
{k,kedar}@research.bell-labs.com

<http://cm.bell-labs.com/cm/cs/who/{k,kedar}>

Abstract. Model checking is an automated approach to the formal verification of hardware and software. To allow model checking tools to be used by the hardware or software designers themselves, instead of by verification experts, the tools should support specification methods that correspond closely to the common usage. For hardware systems, timing diagrams form such a commonly used and visually appealing specification method. In this paper, we introduce a class of synchronous timing diagrams with a syntax and a formal semantics that is close to the informal usage. We present an efficient, compositional algorithm for model checking such timing diagrams. This algorithm has been implemented in a user-friendly tool called RTDT (the Regular Timing Diagram Translator). We have applied this tool to verify several properties of Lucent's PCI synthesizable core.

1 Introduction

Model checking [8,24,9] is a fully automated method for determining whether a hardware or software design, represented as a finite state program, satisfies a temporal correctness property. Currently, many model checking tools are used most effectively by verification experts. In order to make these tools accessible to the hardware or software designers themselves, the tools should support specification methods that correspond closely to common usage. For hardware systems, *timing diagrams* form such a commonly used and visually intuitive specification method. Timing diagrams are, however, often used informally without a well-defined semantics, which makes it difficult, if not impossible, to use them as specifications for formal verification. In this paper, therefore, we precisely define a class of timing diagrams called *Synchronous Regular Timing Diagrams* (SRTD's) and provide a formal semantics that corresponds closely to the informal usage.

A key issue in using timing diagrams for model checking is whether the algorithms that translate timing diagrams into more basic specification formalisms such as temporal logic or ω -automata yield formulas or automata that are of

*** Work supported in part by NSF grant 980-4736 and TARP 003658-0650-1999.

small size. Previous work on model checking for timing diagrams, e.g., with Symbolic Timing Diagrams [10,5,7], with non-regular timing diagrams [12] and with Presburger arithmetic [3] provides algorithms that are, in the worst-case, of exponential or higher complexity in the size of the diagram. Our timing diagram syntax facilitates a decompositional, *polynomial-time* algorithm for model checking. Our experience with verifying Lucent's PCI synthesizable core and other protocols indicates that the SRTD syntax can express common timing properties and is expressive enough for industrial verification needs.

In previous work [1,2], we proposed a class of timing diagrams called RTD's (for Regular Timing Diagrams) that are particularly well-suited for describing *asynchronous* timing, such as that arising, for instance, in asynchronous read/write bus transactions. It is also quite common to have a *synchronous* timing specification, where the changes in values along a signal waveform are tied to the rising or falling edges of a clock waveform. While these specifications can be encoded as RTD's, the encoding introduces a large number of dependency edges between each transition of the clock and each waveform, which results in RTD's that are visually cluttered and have (unnecessarily) increased complexity for model checking. The SRTD notation proposed in this paper is, therefore, tailored towards describing synchronous timing specifications in a visually clean manner. More importantly, we exploit the structure of SRTD's to provide a model checking algorithm that is more efficient than that for RTD's. We present a *decompositional* model checking algorithm that constructs an ω -automaton of size quadratic in the timing diagram size (compared with a cubic size complexity in [2] for RTD's). This automaton, which represents all system computations that *falsify* the diagram specification, is composed with the system model and it is checked if the resulting automaton has an empty language using standard algorithms (cf. [26]). If the language is not empty, there is a system computation that falsifies the specification; otherwise, the system satisfies the specification.

This algorithm is implemented in a tool - the *Regular Timing Diagram Translator* (RTDT). RTDT provides a user-friendly graphical editor for creating and editing SRTD's and a translator that compiles SRTD's to the input language of the formal verification tool COSPAN/FormalCheck [14]. The output of the tool can be easily re-targeted to other verification tools such as SMV [21] and VIS [6]. We used RTDT to verify that Lucent's synthesizable PCI Core satisfies several properties encoded as SRTD's; the SRTD's were formulated by looking at the actual timing diagrams in the PCI Bus specification [23] and the PCI Core User's manual [4].

The rest of the paper is organized as follows. Section 2 presents the syntax and semantics of SRTD's. In Section 3, we describe the decompositional translation algorithm that converts SRTD's into ω -automata. The features of the tool RTDT are described in Section 4. Section 5 illustrates applications of the RTDT tool to a Master-Slave memory access protocol and the synthesizable PCI Core of Lucent's F-Bus. We conclude with a discussion of related work in Section 6.

2 Synchronous Regular Timing Diagrams

A Synchronous Regular Timing Diagram (henceforth referred to as an SRTD or diagram), in its simplest form, is specified by describing a number of waveforms with respect to the clock. A *clock point* is defined as a change in the value of the clock signal. The clock is depicted as waveform defined over $\mathcal{B} = \{0, 1\}$ where the value toggles at consecutive clock points. A *clock cycle* is the period between any two successive rising or falling edges of the clock waveform.

In SRTD's, an *event*, which is a change in the signal value, must occur at either a rising edge of the clock (rising edge triggered) or at a falling edge (falling edge triggered). In the SRTD in Figure 1, signals p and r are falling edge triggered while q is rising edge triggered. Timing diagrams may either be *unambiguous*, where the events are linearly ordered, or *ambiguous*, where the events are partially ordered with respect to time [11]. Synchronous timing diagrams are generally unambiguous but the don't-care transitions do introduce some degree of ambiguity in SRTD's.

2.1 Syntax

In most applications of timing diagrams, the waveform behavior specified by the diagram must hold of a system only after a certain *precondition* holds. This condition may be a boolean condition on the values of one or more signals (a *state* condition), or a condition on the signal values over a finite period of time (a *path* condition). To accommodate this type of reasoning, we permit the more general form of path preconditions to be specified in an SRTD. Preconditions are specified graphically by a solid vertical marker that partitions the SRTD into two disjoint parts, a precondition part that includes all the events at and to the left of the marker and a postcondition part that contains all the events to the right of the marker. The precondition of the diagram in Figure 1 is a path precondition, given by the path $\langle \bar{p}(q + \bar{q})r \rangle; \langle \bar{p}(q + \bar{q})\bar{r} \rangle; \langle \bar{p}\bar{q}\bar{r} \rangle$ (the angle brackets indicate the constraints on signal values at a clock edge, while “;” indicates succession in time, measured by clock edges).

A common feature in synchronous timing diagrams is a way to express that the value of a signal during a certain period is not important. We use *don't-care values* to specify that the value at a point is unknown, unspecified or unimportant. In Figure 1, the don't-care values on waveform q are used to state that q should not be considered in the precondition. With the addition of preconditions, one can express properties of the form “if B rises then A rises in exactly 5 time units”. In order to specify richer properties such as “if B rises then A rises within 5 time units”, we need a way of stating that the exact occurrence of the rising transition of A is not important as long as it is within the specified time bound. In SRTD's, we use a *don't-care transition* to graphically represent this temporal ambiguity. The don't-care transition is defined for a particular waveform over one or more clock cycles; its semantics specifies that the signal may change its value at any time during the specified interval and that, once it changes, it remains stable for the remainder of the interval. This stability requirement is the

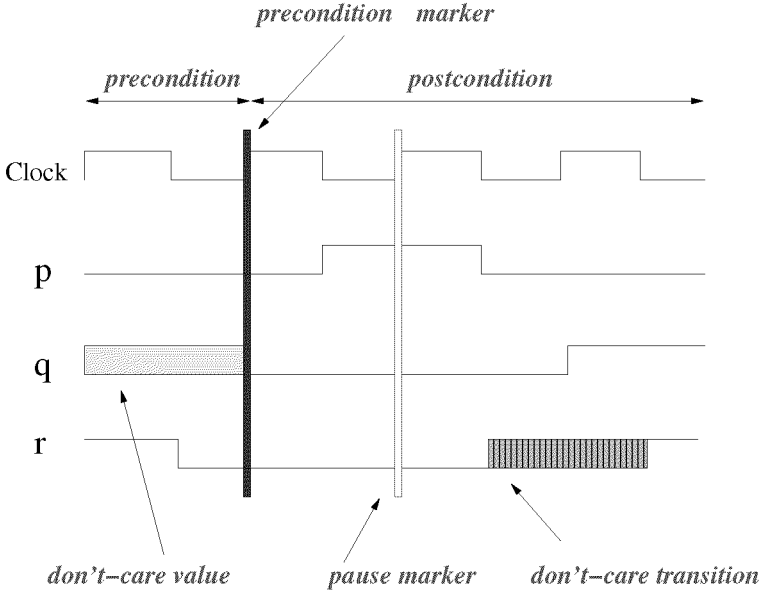


Fig. 1. Annotated Synchronous Regular Timing Diagram

only difference between don't-care transitions and don't-care values. In Figure 1, the don't-care transition allows signal r to rise in either the third or fourth clock cycle.

In addition, in loosely coupled systems, it may not be always necessary to explicitly tie every event to the clock. This is useful in stating eventuality properties like “every memory request is eventually followed by a grant”, and is represented diagrammatically by a pause marker. A *pause* specifies that there is a break in explicit timing at that point, i.e. the state of the signals (except the clock) remains unchanged (stutters) for an arbitrary finite period of time before changing. In Figure 1 the pause at the end of the second clock cycle indicates that the state $\langle p\bar{q}\bar{r} \rangle$ stutters for a finite period (until p changes at a falling edge). The pauses allow us to express richer properties like “if A rises then eventually B rises”.

We have observed that, in practice, both pauses and don't-care objects occur in timing diagrams, and that preconditions are often implicit in the assumptions that are made with respect to when a diagram must be satisfied. In reviewing many specifications and from our discussion with engineers, we are led to believe that SRTD's correspond closely to informal usage and are expressive enough for industrial verification needs.

We now define SRTD's formally. An SRTD is defined over a set of “symbolic values” $\mathcal{SV} = \mathcal{B} \cup \{X, D\}$, where X is a don't-care value and D indicates a don't-care transition. The set \mathcal{SV} is ordered by \sqsubseteq , where $a \sqsubseteq b$ iff either $a=b$

or $a \in \{X, D\}$ and $b \in \mathcal{B}$. The alphabet of an SRTD defined over n signals is $\mathcal{SV}_n = \{(a_1 a_2 \dots a_n) \mid a_1 \in \mathcal{SV} \wedge \dots \wedge a_n \in \mathcal{SV}\}$. Here, we have taken the set of defined values to be the boolean set \mathcal{B} but our algorithms and results also apply when this is any fixed finite set, such as an enumeration of the possible values of a multi-valued signal.

Definition 1 (SRTD). *An SRTD T is a tuple (c, S, WF, M) where*

- $c > 1$ is an integer that denotes the number of clock points.
- S is a non-empty set of signal names (excluding the clock).
- WF is a collection of waveforms; for each signal $A \in S$, its associated waveform is a function $WF_A : [0, c) \rightarrow \mathcal{SV}$, while the associated waveform for the clock is $WF_{clk} : [0, c) \rightarrow \mathcal{B}$.
- M is a finite (non-empty) ascending sequence $0 \leq M_0 < M_1 < \dots < M_{k-1} < c - 1$ of position markers. M_0 is the precondition marker, while for each $i > 0$, M_i is the i -th pause marker.

To facilitate defining the semantics as well as the algorithms it is also helpful to view an SRTD as a collection of *segments*, where each segment is essentially a vertical slice of the timing diagram, encompassing all waveforms between two successive markers or a marker and the start/end of the diagram. The k markers in M partition the interval $[0, c)$ in an SRTD T into $k + 1$ disjoint sub-intervals $I_0 = [0, M_0]$, $I_1 = (M_0, M_1]$, ..., $I_{k-1} = (M_{k-2}, M_{k-1}]$, $I_k = (M_{k-1}, c - 1]$. The length m_0 of the interval I_0 is $M_0 + 1$, while for intervals I_i , with $i \in [1, k)$, the length m_i of I_i is $M_i - M_{i-1}$, and the length of the last interval I_k is $c - 1 - M_{k-1}$. The k markers, therefore, partition an SRTD into $k + 1$ segments.

Definition 2 (Segment). *The segment Seg_i ($i \in [0, k]$) that corresponds to the interval I_i of length m_i is defined to be a function $Seg_i : S \times [0, m_i) \rightarrow \mathcal{SV}$, where for each $j \in [0, m_i)$ and $A \in S$, $Seg_i(A)(j) = WF_A(j)$ when $i = 0$ and $Seg_i(A)(j) = WF_A(M_{i-1} + 1 + j)$ when $i > 0$.*

Any SRTD $T = (c, S, WF, M)$ can be represented as the tuple of segments $(Pre, Post_1, \dots, Post_k)$ as defined above. Segment Pre (Seg_0) represents the precondition, while segments $Post_i$ (Seg_i), for $i > 0$, represent successive postcondition segments. For instance, the SRTD in Figure 1 has three segments, one precondition segment and two postcondition segments. For each signal A , $Seg_i(A)$ is a function from $[0, m_i) \rightarrow \mathcal{SV}$ which describes the waveform for signal A in the i th segment. This representation of an SRTD is useful in the sequel.

We impose certain well-formedness criteria on SRTD's. In preparation, we define an event to be *precisely locatable* if it occurs at a clock point where the signal value changes from 0 to 1 or vice versa. In Figure 1, the falling edge of waveform p in the third clock cycle is precisely locatable while the don't care transition in waveform r is not a precisely locatable event.

Definition 3 (Well-Formed SRTD). An SRTD $T = (Pre, Post_1, \dots, Post_k)$ is well-formed iff

- The precondition segment Pre does not have any don't-care transitions¹. Note that the precondition can have don't-care values.
- There is at least one precisely locatable event in the clock cycle immediately following each pause.
- Any maximal non-empty sequence of D 's (don't-care transitions) must be immediately preceded by a boolean value and followed by the negation of this value.
- Every event in a waveform designated as rising(falling) edge triggered must occur at a rising(falling) edge of the clock.

2.2 Semantics

An SRTD defines properties of *computations*, which are sequences of *states*, where a state is an assignment of values from \mathcal{B} to each of the n waveform signals. A computation is defined over the alphabet $\mathcal{B}_n = \{(a_1 a_2 \dots a_n) | a_1 \in \mathcal{B} \wedge \dots \wedge a_n \in \mathcal{B}\}$. For any computation y , we use y_A to denote the projection of y on to the coordinate for signal A .

Definition 4 ($\dot{\subseteq}$). For a finite waveform segment $Seg_i(A) : [0, m_i) \rightarrow \mathcal{SV}$ and a projection y_A of computation y with length m_i ($y_A \in \mathcal{B}^{m_i}$), $Seg_i(A) \dot{\subseteq} y_A$ iff with length m_i

- For every $p \in [0, m_i)$, $Seg_i(A)(p) \subseteq y_A(p)$.
- For every p, q , if $Seg_i(A)[p..q]$ has the form $(a; D^+; \bar{a})$ then $y_A[p..q]$ has the form $(a^+; \bar{a}^+)$, where $a, \bar{a} \in \mathcal{B}$ and $\bar{a} \neq a$.

Definition 5 (Segment Consistency). A segment Seg_i of length m_i is satisfied by a sequence $y \in \mathcal{B}_n^{m_i}$ iff for each signal A , $Seg_i(A) \dot{\subseteq} y_A$ holds.

We will now construct regular expressions for the precondition Pre_T and the postcondition $Post_T$ of a SRTD T . By the definition above of segment consistency, any Pre or $Post_i$ segment can be represented as an extended regular expression of the form $\bigwedge_{s \in S} r_s$, where r_s encodes the constraints for the waveform for signal s in the segment. The regular expression for $Post_T$ is the concatenation of sub-expressions that correspond to each $Post_i$ segment separated by an expression for each pause. Thus, $Post_T = (seg_1; val_1^*; seg_2; val_2^*; \dots; seg_{k-1})$, where seg_i is the regular expression for segment $Post_i$ and val_i is the vector of values at the last position ($m_i - 1$) in $Post_i$, which is at the pause marker separating it from $Post_{i+1}$.

Definition 6 (Always Followed-By). $G(p \hookrightarrow q)$ holds of a computation σ iff, for all i, j such that $j \geq i$, if sub-computation $\sigma[i \dots j] \models p$, then there exists k such that $\sigma[j + 1 \dots k] \models q$.

¹ We can relax this requirement and our translation algorithm is still applicable. In that case, however, we cannot guarantee an efficient translation.

In the definition above, p and q are arbitrary path properties; however, when p is a state property, $\mathbf{G}(p \hookrightarrow q)$ is equivalent to $\mathbf{G}(p \Rightarrow \mathbf{X}q)$, where \mathbf{X} is the next time operator. An infinite computation σ satisfies an SRTD T (written $\sigma \models T$) if and only if every finite segment that satisfies the precondition is immediately followed by a segment that satisfies the postcondition of the diagram. This is formalized in Definition 7.

Definition 7 (SRTD Semantics). *An infinite computation σ satisfies an SRTD T ($\sigma \models T$) iff $\sigma \models \mathbf{G}(Pre_T \hookrightarrow Post_T)$.*

3 Model Checking SRTD's

We first present an algorithm that translates an SRTD into an ω -automaton for the *negation* of the SRTD property. We then present the model checking algorithm that makes use of this automaton.

3.1 Translation Algorithm

The algorithm translates SRTD's into ω -automata, which are finite state automata accepting infinite computations as input (cf. [17]). It proceeds by decomposing T into waveforms and producing sub-automata that track portions of each waveform. It consists of the following four steps.

1. Partition the diagram into the precondition part and the postcondition part.
2. Construct a single deterministic automaton \mathcal{A}_P for the precondition. This automaton tracks the values of all signals simultaneously over the number of clock cycles of the precondition. Since the precondition cannot contain don't-care transitions, this automaton has linearly many states in the length of the precondition.
3. Construct a deterministic automaton S_i for each signal i of the postcondition. The automaton S_i tracks the waveform for signal i over all the postcondition segments. The automaton checks at each clock cycle that the waveform has the specified value. For a don't-care transition, the automaton maintains an extra bit that records whether the transition has occurred. For a pause, the automaton goes into a "waiting" state, which it leaves when the precisely locatable (non-don't-care) event signaling the end of the pause occurs. The number of states of this automaton is thus linear in the length of the postcondition. In our model, the pause condition is required to hold for only a finite (but unbounded) number of cycles. Thus, S_i has a fairness condition which ensures that the automaton does not stay in a waiting state forever.
4. Construct an NFA $\mathcal{A}_{\bar{T}}$ for the negation of the SRTD property of T that operates as follows on an infinite input sequence: it nondeterministically "chooses" a point where the precondition holds, runs the DFA \mathcal{A}_P at this point and if \mathcal{A}_P accepts it then "chooses" a postcondition DFA S_i and runs this automaton at the point where \mathcal{A}_P accepted and accepts if this automaton rejects.

A $\forall FA$ [20,25] is a finite state automaton that accepts an input iff *every* run of the automaton along the input meets the acceptance criterion. An SRTD T can be represented succinctly by a $\forall FA$ \mathcal{A}_T that has the identical structure as the NFA $\mathcal{A}_{\bar{T}}$ but with a complemented acceptance condition.

The *size* of an SRTD is the product of the number of signals and the number of clock cycles. The number of clock cycles does not include the indeterminate amount of time represented by a pause; it refers only to the explicitly indicated clock cycles in the diagram. The automata produced by the translation algorithm all have linearly many states, in terms of the size of the SRTD.

Theorem 1. (Correctness) *For any SRTD T and $x \in \mathcal{B}_n^\omega$, $x \models T$ iff $x \in L(\mathcal{A}_T)$.*

Theorem 2. (Complexity) *For any SRTD T and the equivalent $\forall FA$ \mathcal{A}_T , the size of \mathcal{A}_T is quadratic in $|T|$.*

Proof. The size of an SRTD $T = (Pre, Post_1, \dots, Post_k)$ is $n * c$, where n is the number of waveforms and c is the number of clock points. We assume that the transitions in \mathcal{A}_T are labeled with boolean formulas over the n signals. The size of the transitions in \mathcal{A}_T is the sum of the length of the formulas labeling the transitions. The size of \mathcal{A}_T is $s + t$, where s is the number of states and t is the transition size.

The number of states s in the monolithic automaton for the precondition \mathcal{A}_P , is bounded by the number of clock points in the precondition, therefore $s < c$. Since each transition encodes the values of the signals at each point, the size of each transition is $O(n)$ and the number of such transitions is bounded by c . Thus, the transition size is linear in $|T|$.

The number of states s in S_i is bounded by the number of clock points c , therefore $s \leq c$. Except for the pause transition, the transitions are labeled with constant size formulae. The pause transition may be dependent on a number of (simultaneous) signal value changes, so it can have size at most n . Thus, the overall transition size for S_i is of order $|T|$; hence, S_i has size linear in the size of T .

The size of the $\forall FA$ \mathcal{A}_T is the sum of the sizes of the precondition and the n postcondition automata and is thus $(n + 1) \cdot |T| = O(|T|^2)$.

□

3.2 Model Checking

Theorem 2 shows that an SRTD property can be represented succinctly by a $\forall FA$. A *monolithic* translation of the property yields an NFA that requires a postcondition automaton that is essentially the product (intersection) of all the S_i automata. This monolithic NFA can be of size exponential in the size of the SRTD, because it needs to take into account all possible interleavings of the don't-care transitions of the postcondition.

Recall that the property represented by the SRTD T is $\mathbf{G}(Pre_T \hookrightarrow Post_T)$. Since $Post_T = \bigwedge_i S_i$, this property can be decomposed into the conjunction of

individual checks $\mathbf{G}(Pre_T \hookrightarrow S_i)$. In a typical model checker, this check is performed by determining if there is a computation of the system that satisfies the *negation* of the property. The check can be done by determining if there is a path to a point where \mathcal{A}_P accepts, followed by a computation where S_i rejects. Since S_i is a DFA, it can be complemented to form an automaton of the same size. Hence, model-checking can be done efficiently with this decomposed representation of the postcondition. A similar observation was made for the analysis of asynchronous timing diagrams in [2].

Theorem 3. *For a transition system M and an SRTD T , the time complexity of model checking is linear in the size of M and quadratic in the size of T .*

Proof. The $\forall FA \mathcal{A}_T$, corresponding to T , is the automaton for $\mathbf{G}(Pre_T \hookrightarrow \bigwedge_i S_i)$ where Pre_T is the automaton for Pre and each S_i is the automaton for the postcondition segment of waveform i . The problem of checking $M \models \mathcal{A}_T$ can be decomposed into $\bigwedge_i M \models \mathcal{A}_i$, where \mathcal{A}_i is the automaton for $\mathbf{G}(Pre_T \hookrightarrow S_i)$. We can check $M \models \mathcal{A}_i$ in time linear in the size of M and \mathcal{A}_i , which by Theorem 2 is $O(|M|.|T|)$. But we have $|S|$ such verification tasks, thus the time complexity of checking $M \models \mathcal{A}_T$ is $O(|M|.|T|^2)$.

□

4 The RTDT Tool

RTDT is a tool that translates SRTD's into ω -automata definitions which are input to the verification tool COSPAN [14]. The tool has an editor to create SRTD's and a translator that outputs the corresponding descriptions in COSPAN's input language.

The RTDT editor is a graphical environment, written in Java, that enables a user to create and edit SRTD's. The editor is almost entirely mouse driven. There are options to open, save and print existing SRTD descriptions. A user may easily add or delete waveforms, clock cycles and pauses. The precondition defaults to the initial clock point but the user can set the precondition to a path condition. Editing a waveform is done by positioning the mouse on the waveform and clicking either the left or right button. The editor is designed to ensure that the diagrams created are well-formed SRTD's by construction. The tool provides a user-friendly interface for specifying SRTD's. Figure 2 is a screen shot of the interface.

The output of the tool is currently targeted to the formal verification tool COSPAN/FormalCheck [14]. We use a macro in COSPAN/FormalCheck called a *strobe* that recognizes a specified waveform. The RTDT translator automatically generates strobe definitions for the diagram using the algorithm outlined in Section 3. The resulting strobe definition can be viewed through the editor as in Figure 2 or saved in a file to be used as the specification in model checking the system under verification.

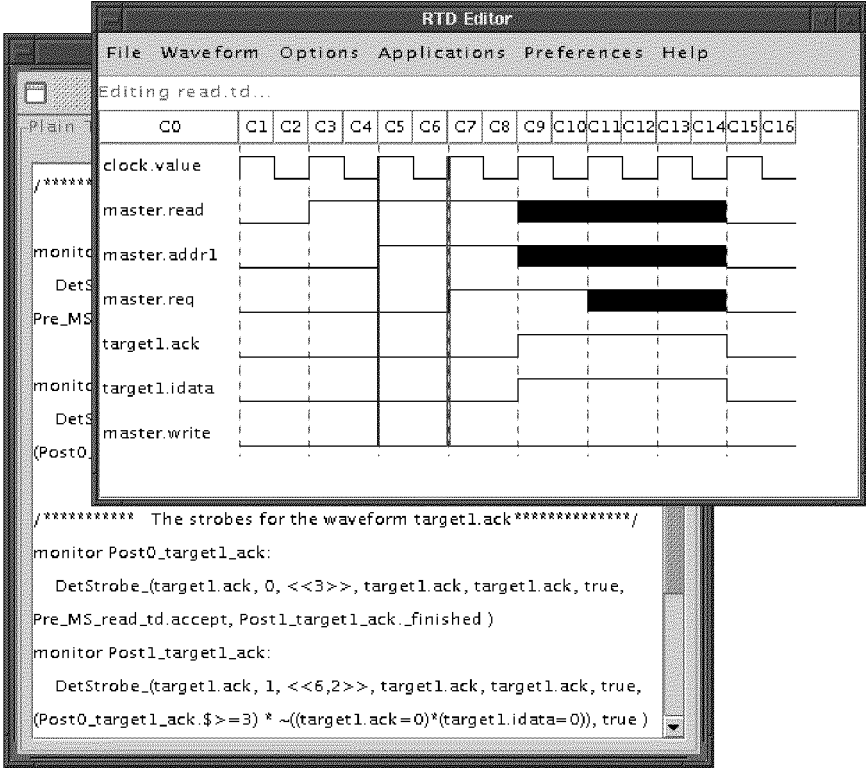


Fig. 2. Editing and Viewing Screens of RTDT

RTDT gives the user the option of invoking COSPAN/FormalCheck from within the tool. When a property fails to hold of a system then COSPAN/FormalCheck generates a failure trace. This trace corresponds to a bad path through the system and is usually long and very hard to read. Currently many model checkers display these traces graphically as synchronous timing diagrams. RTDT offers the added advantage of allowing the user to edit these error traces. One can remove irrelevant signals and clock cycles. Furthermore the precondition may be used to direct attention to erroneous regions of the design. This feature allows the use of a relevant portion of the trace as a new property; this is useful in debugging the system.

Although the output of the tool is currently targeted to COSPAN/FormalCheck, with very little effort, the tool can be re-targeted to generate output suitable for other formal verification tools such as SMV [21] or VIS [6].

5 Applications

The true test of the efficiency of our algorithms is how they fare in practice on industrial examples of all sizes. Towards this end, we used RTDT with COSPAN/FormalCheck to verify two systems. The first is a synchronous master-slave memory system and the second is the synthesizable Core of Lucent's F-Bus.

5.1 Master-Slave Memory System

The Master-Slave memory system consists of one master module and three slave modules. In the master-slave system, the master issues a memory instruction and the slaves respond by accessing memory and performing the operation. The master initiates the start of a transaction by asserting either the read or write line. Next the master puts the address on the address bus and asserts the *req* signal. The slave whose tag matches the address awakens, services the request, then asserts the *ack* line on completion. Upon receiving the *ack* signal the master resets the *req* signal, causing the slave to reset the *ack* signal. Finally, the master resets the address and data buses.

Table 1. Verification Statistics for Master-Slave Design

Design	Number of BDD variables	Average BDD size	Average Space (MBytes)	Average Time (seconds)
master-slave	67	11405	0.85	–
read (C)	95	13433	0.86	0.32
read (M)	205	22079	1.46	3.19
write (C)	95	11542	0.86	0.31
write (M)	205	21915	1.45	2.51

We verified that this system satisfied both read (see Figure 2) and write memory transactions formulated as SRTD's. The SRTD's were created with the RTDT editor and the translator was used to generate the corresponding COSPAN/FormalCheck descriptions. We used COSPAN/FormalCheck to model check the system with respect to these descriptions.

Recall that a monolithic translation of an SRTD yields an *NFA* that is essentially the product (intersection) of the *DFA*'s for each waveform. In order to compare our decompositional algorithms with monolithic algorithms, we did the verification checks both decompositionally and monolithically. In Table 1, *read(M)* corresponds to the verification check on the master-slave design and

the monolithic automaton for the read SRTD while $read(C)$ corresponds to the verification check done on the master-slave design and automata for a single waveform. The numbers in Table 1 for BDD size, space and time for the decomposition check is the average over the individual verification checks for each waveform. For example, the total amount of time taken to verify the $read$ SRTD decompositionally was 3.23 seconds and this is a little more than the time taken for the single monolithic verification. Our verification numbers show that the decompositional checks consistently use less space while generally taking more time. Notwithstanding the Lichtenstein-Pnueli thesis [18], in practice, as one reaches the space limitations of symbolic model checking tools, efficiency with respect to space is of more importance. We observe that the decompositional check, with respect to BDD size and space, is not much larger than the size of the system itself. The monolithic verification is, however, significantly more expensive.

5.2 Lucent's PCI Synthesizable Core

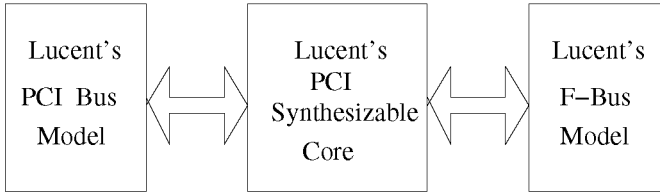


Fig. 3. Block Diagram of Lucent's F-Bus with PCI Core

The PCI Local Bus is a high performance, 32-bit or 64-bit bus with multiplexed data and address lines, which is now an industry standard. The PCI bus is used as an interconnect mechanism between processor/memory systems and peripheral controller components. Lucent Technologies' PCI Interface Synthesizable Core is a set of synthesizable building blocks that designers can use to implement a complete PCI interface. The PCI Interface Synthesizable Core is designed to be fully compatible with the PCI Local Bus specification [23]. The Synthesizable Core bridges an industrial standard PCI bus to an F-Bus, which is 32-bit internal buffered FIFO bus that supports a Master-slave architecture with multiple masters and slaves.

We used Lucent's PCI Bus Functional Model shown in Figure 3, which is a sophisticated simulation environment that was developed to test the Synthesizable Core for functionality and compliance with the PCI specification [23]. The Functional Model consists of the PCI Core blocks and abstract models for both the PCI Bus and the F-Bus. The PCI Bus and F-Bus models were designed to fully exercise the PCI Synthesizable Core in both the slave and master

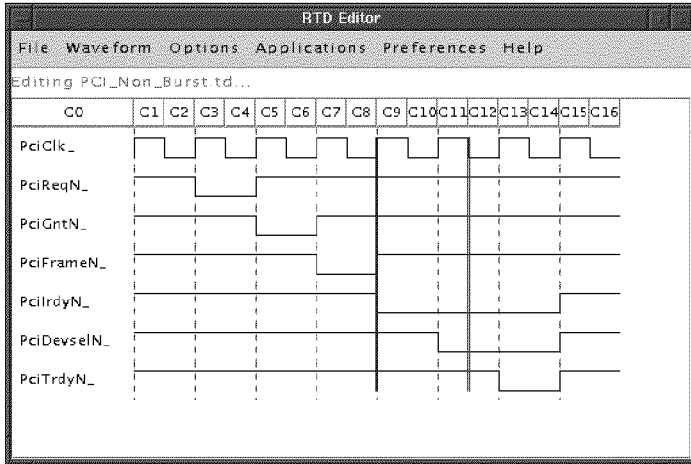


Fig. 4. SRTD for the Non Burst Transaction of the PCI Bus

modes. This model has about 1500 bounded state variables and was too large for model checking. We had to perform some abstractions, like freeing variables and removing variables from consideration for cone of influence reductions. These abstractions were property-specific and had to be modified for each property checked.

The Synthesizable Core design is synchronous to the PCI clock. The basic bus transfer on the PCI is a burst, which is composed of an address phase followed by one or more data phases. In the non-burst mode, each address phase is followed by exactly one data phase. The data transfers in the PCI protocol are controlled by three signals *PciFrame*, *PciIrdy* and *PciTrdy*. The master of the bus drives the signal *PciFrame* to indicate the beginning and end of a transaction. *PciIrdy* is asserted by the master to indicate that it is ready to transfer data. Similarly the target uses *PciTrdy* to signal that it is ready for data transfer. Data is transferred between master and target on each rising clock edge for which both *PciIrdy* and *PciTrdy* are asserted. We verified that the PCI Core satisfied several timing diagram properties for both the burst and non-burst modes. We formulated the properties as SRTD's by looking at the actual timing diagrams that occurred in the PCI specification [23] and the PCI Core User's Manual [4]. Figure 4 is one of the properties that we verified for the non burst mode.

The verification was done both monolithically and compositionally and Table 2 presents the verifications statistics. In Table 2, the size, space and time

Table 2. Verification Statistics for Lucent’s Synthesizable PCI Core

Design	Number BDD variables	Average BDD size	Average Space (MBytes)	Average Time (seconds)
PCI Prop1 (M)	740	715157	36.2	411
PCI Prop1 (C)	664	417816	22.1	279
PCI Prop2 (M)	1036	688424	23.9	209
PCI Prop2 (C)	996	554866	19.1	182
PCI Prop3 (M)	749	3742074	198.6	16793
PCI Prop3 (C)	699	2680421	171.7	5677

numbers for properties with the suffix *(M)* correspond to the verification check on the abstracted PCI Core and the monolithic automaton for the property. The suffix *(C)* refers to the average over the individual decompositional verification checks on the abstracted system and the automata for each waveform. Table 2 shows a savings of up to 30% in BDD size and corresponding savings in space. In practice, as one reaches the space bounds of a model checking tool, it may be beneficial to trade time for space. Our results demonstrate that our decompositional approach is more space efficient than a monolithic one.

6 Related Work and Conclusions

Various researchers have investigated the formal use of timing diagrams. A verification environment for embedded systems, called *SACRES* [5,7], allows users to graphically specify properties as Symbolic Timing Diagrams (STD’s) [10]. STD’s are, however, asynchronous in nature and cannot explicitly tie events to the clock. Moreover, the translation algorithm is monolithic, and in general results in an exponential translation. Fisler [12] provides a procedure to decide regular language containment of non-regular timing diagrams, but the model checking algorithms have a high complexity (PSPACE). Cerny et al. present a procedure [16] for verifying whether the finite behavior of a set of action diagrams (timing diagrams) is consistent. Amon et al. [3] uses Presburger formulas to determine whether the delays and guarantees of an implementation satisfy constraints specified as a timing diagram. This work uses Timing Designer² to specify the constraints and delays. They have developed tools that generate Presburger formulas corresponding to the timing diagrams and manipulate them.

² Timing Designer is a commercial timing diagram editor from Chronology Corporation.

This model cannot, however, handle synchronous signals, and the algorithm for verifying Presburger formulas is multi-exponential in the worst case.

In contrast, for SRTD's, we have presented a decompositional, efficient algorithm for model checking, which has time complexity that is linear in the size of the system model and quadratic in the size of SRTD. Our experience with verifying the PCI core and other protocols indicates that the syntax of SRTD's suffices to express common timing properties, and is expressive enough for industrial verification needs.

We have also developed a tool, RTDT, which implements the translation from SRTD's to ω -automata and have used it to verify several timing specifications from the PCI specification for Lucent's Synthesizable Core. The output of RTDT is currently targeted to the COSPAN/FormalCheck tool, but it can be easily re-targeted to produce output suitable for other model checkers (e.g. SMV, VIS). In addition to editing and translating SRTD's, the tool allows the user to view error traces and convert these into timing properties. There are other timing diagram editors [19,13,15,22] which employ the timing specifications for test generation, simulation or synthesis but they do not, to the best of our knowledge, have a formal verification capability.

Acknowledgments

We would like to thank Vanya Amla, Pankaj Chauhan and Phoebe Weidmann for helpful discussions.

References

1. N. Amla and E.A. Emerson. Regular Timing Diagrams. In *LICS Workshop on Logic and Diagrammatic Information*, June 1998.
2. N. Amla, E.A. Emerson, and K.S. Namjoshi. Efficient Decompositional Model Checking for Regular Timing Diagrams. In *CHARME*. Springer-Verlag, September 1999.
3. T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic Timing Verification of Timing Diagrams Using Presburger Formulas. In *DAC*, 1997.
4. Bell Laboratories, Lucent Technologies. PCI Core User's Manual (Version 1.0). Technical report, July 1996.
5. A. Benveniste. Safety Critical Embedded Systems Design: the SACRES approach. Technical report, INRIA, May 1998. URL: <http://www.tni.fr/sacres/index.html>.
6. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS. In *FMCAD*, 1996.
7. U. Brockmeyer and G. Wittich. Tamagotchis need not die-Verification of STATE-MATE Designs. In *TACAS*. Springer-Verlag, March 1998.
8. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logics of Programs*, volume 131. Springer Verlag, 1981.

9. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
10. W. Damm, B. Josko, and Rainer Schölör. Specification and Verification of VHDL-based System-level Hardware Designs. In Egon Borger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
11. K. Fisler. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams*. PhD thesis, Computer Science Department, Indiana University, August 1996.
12. K. Fisler. Containment of Regular Languages in Non-Regular Timing Diagrams Languages is Decidable. In *CAV*. Springer Verlag, 1997.
13. W. Grass, C. Grobe, S. Lenk, W. Tiedemann, C.D. Kloos, A. Marin, and T. Robles. Transformation of Timing Diagram Specifications into VHDL Code. In *Conference on Hardware Description Languages*, 1995.
14. R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102, 1996.
15. K. Kastein and M. McClure. Timing Designer use for Interface Verification at Symbios Logic. *Integrated System Design*, May 1997. URL: <http://www.chronology.com>.
16. K. Khordoc and E. Cerny. Semantics and Verification of Timing Diagrams with Linear Timing Constraints. *ACM Transactions on Design Automation of Electronic Systems*, 3(1), 1998.
17. R.P. Kurshan. *Computer-aided verification of coordinating processes: the Automata-theoretic approach*. Princeton University Press, 1994.
18. O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs satisfy their Linear Specifications. In *POPL*, 1985.
19. K. Luth. The ICOS Synthesis Environment. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1998.
20. Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs by \forall -Automata. In *POPL*, 1987.
21. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
22. D. Mitchell. Test Bench Generation from Timing Diagrams. In David Pellerin, editor, *VHDL Made Easy*. 1996. URL: <http://www.syncad.com>.
23. PCI Special Interest Group. PCI Local Bus Specification Rev 2.1. Technical report, June 1995.
24. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.
25. M. Vardi. Verification of Concurrent Programs. In *POPL*, 1987.
26. M. Vardi and P. Wolper. An Automata-theoretic Approach to Automatic Program Verification. In *LICS*, 1986.

Model Reductions and a Case Study

Jin Hou and Eduard Cerny

Dépt.IRO, Université de Montréal, Montréal, QC, Canada

hou@iro.umontreal.ca, cerny@iro.umontreal.ca

Abstract. In this paper, we present a model reduction algorithm for property checking. For the property to be verified, we first construct a *property dependency graph* which represents the function dependency of the property on variables. Beginning from the set of state variables appearing in the property, we search through the property dependency graph and add a *noncorrelated* set of state variables to the current set of state variables to construct a more detailed model at each reduction iteration step. The final reduced model is the one which is constructed by using all state variables that can be reached in the graph. The final reduced model preserves the property strongly, while the intermediate reduced models preserve the property weakly. Our reduction algorithm is completely automatic. Since there is no preimage operation in MDG (Multiway Decision Graph) model checker due to the presence of abstract state variables, all backward reduction algorithms cannot be used in MDG. Our method is suitable for MDG and has been implemented in this tool, however, it can be used in other tools as well. We then discuss a quite common circuit structure which appears in telecommunication and data processing circuits. We use three verification tools MDG, FormalCheck and SMV to verify this circuit. The experimental results show that our reduction algorithm is more efficient on these typical structures.

1 Introduction

As the complexity of microelectronics systems increases rapidly, the conventional simulation is time consuming and exhaustive simulation is impossible. Interest in formal verification is growing in industry. In the formal verification method of model checking, to verify that an implementation satisfies a specification is based on exploring the reachable state space of its model. The main problem of model checking is the state space explosion, the representation of the state space during verification needs more than the available computer memory. To handle this problem, an explicit representation of the state space which is exponential with the number of state components in a circuit is substituted by characteristic boolean functions to encode sets of states and transition relations. This is referred to as symbolic methods [8, 9, 10]. In particular, ROBDDs [7] have been adopted in SMV and FormalCheck to represent transition/output relations and sets of states. For verifying interactions between the datapaths and control of circuits with large data paths, ROBDD may not be efficient. Multiway Decision Graphs (MDGs) developed by Corella, Cerny et al. [1, 2] can represent the relations and sets of states of an abstract

state machine (ASM). There are variables of abstract sort and uninterpreted functions in MDG. A data signal can be represented as a single abstract variable, which is sufficient for verifying interactions between the datapath and the control. The MDG data structure is used in our verification tool.

Although symbolic methods have significantly increased the useful domain of model checking, the achievable state space is still too small for many realistic systems. What we need is model reductions: if we can reduce model M to a smaller M' such that $M' \models P \Leftrightarrow M \models P$, then property checking can be done by using only M' , which may often avoid state explosion. The relation $M' \models P \Leftrightarrow M \models P$ is referred to as strong preservation of P , and the implication $M' \models P \Rightarrow M \models P$ is referred to as weak preservation of P . One reduction method is to exploit symmetry in the structure of the system [13, 14]. The structural symmetry induces an equivalence relation between states. For verifying the equivalence classes, we need to explore only one state per each class. In [13] the symmetry of a finite state system was formally described, and the conclusion was proved that a formula in CTL* is preserved if all atomic propositions in the formula are invariant under the symmetry group. In [14] the scalarset data type was added to the Mur ϕ description language for specifying symmetry. It is used to eliminate symmetrically selectable registers or individual bits in a word. Symmetry reduction is also used in SMV. Another method is a homomorphic reduction in language containment tests [15, 16, 17], which is implemented in COSPAN (FormalCheck). To verify that a system satisfies a given property is to test whether the ω -regular language associated with the reduced system is contained in the language associated with the property. The reduced automata are derived from the original ones through co-linear automaton homomorphisms.

Since there is abstract data representation in MDG, the bit symmetry reduction is not needed. There is no preimage operation in MDG due to the presence of abstract state variables. Then all backward reduction algorithms [23, 24, 25, 26] cannot be applied to MDG. We present in this paper an iterative reduction algorithm which is suitable for MDG model checker [3, 4], however, it can be used in ROBDD-based tools as well. Our reduction algorithm computes the transition system by using only the individual transition relations of the so-called property dependent state variables VP of an ACTL property P to be verified. It can be easily shown that the resulting abstract system preserves P strongly, and consequently the abstract system constructed using only a subset of VP preserves P weakly. The critical aspect is how to select the subset of VP. The main contribution of this paper is an heuristic iterative reduction algorithm for property checking. We can view the dependency of the property on the determining variables as a directed graph called property dependency graph. The algorithm begins with only the state variables appearing in the property, and then search through the graph to progressively add state variables from VP to construct a reduced system and to check P . We have implemented our algorithm in MDG model checker.

In this paper we also present a case study of a circuit structure which is widely used in telecommunication and data processing digital circuits. We used SMV, FormalCheck and MDG to verify it. The experimental results are presented and compared, which shows that our method can find the appropriate model reduction.

The rest of the paper is organized as follows. In Section 2 we introduce model checking of temporal logic. In Section 3 we define the property dependent state variables VP and noncorrelated sets. In Section 4 we present an iterative algorithm for model reductions. In Section 5 we discuss a common circuit structure and the verification results using MDG, FormalCheck and SMV. Section 6 contains some conclusions.

2 Model Checking of Temporal Logic

A digital circuit can be represented by a labelled transition system $M = \langle S, S_0, I, T, AP, L \rangle$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, I is a finite set of inputs, $T: S \times I \rightarrow S$ is a transition relation, AP is the set of atomic propositions, $L: S \rightarrow 2^{AP}$ is a labelling function indicating which propositions are true in each state. The specification that should be satisfied is a set of properties and the properties are expressed in a temporal logic. Computation Tree Logic (CTL) is widely used to express properties in model checking tools (SMV, etc.). CTL has path quantifiers A (“for all computation paths”) and E (“for some computation paths”) as well as linear-time operators G (“always”), F (“sometimes”), X (“next-time”), and U (“until”). The linear-time operators must be immediately preceded by a path quantifier. A path is defined as an infinite sequence of states, with each adjacent pair related by T . Every atomic proposition is a CTL formula. If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, AXf , EXf , $A(fUg)$, $E(fUg)$. The other CTL formulas $f \vee g$, AFf , EFf , AGf , EGf can be derived. The truth of a CTL formula is defined inductively [6].

In MDG model checking, the design is represented by an Abstract State Machine (ASM) [1, 2], the properties to be verified are expressed by formulas in the first-order ACTL temporal logic L_{MDG} [3, 4]. An ASM defined here is a state machine in which input, state and output variables can be of an abstract sort, and operations can be uninterpreted function symbols, and MDGs are used to represent sets of states, transition and output relations. The formal logic underlying MDGs is a many-sorted first-order logic with the distinction between abstract sorts and concrete sorts. Concrete sorts have enumerations of individual constants, while abstract sorts do not. Due to this distinction, there are concrete variables and abstract variables, individual constants which appear in enumerations and generic constants of abstract sorts. Let f be a function symbol of type $\alpha_1 \times \dots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an abstract function symbol. If all the $\alpha_1 \dots \alpha_{n+1}$ are concrete, f is a concrete function symbol. If α_{n+1} is concrete and at least one of $\alpha_1 \dots \alpha_n$ is abstract, then f is a cross-operator. An equation is an expression “ $A_1 = A_2$ ” where A_1 and A_2 are terms of the same sort. Atomic formulas are the equations, plus T (truth) and F (falsity). Formulas are built from the atomic formulas in the usual way using logical connectives and quantifiers. A multiway decision graph (MDG) is a finite directed acyclic graph where the leaf nodes are labelled by formulas, the internal nodes are labelled by terms, and the edges issuing from an internal node N are labelled by terms of the same sort as the label of N . Such a graph G represents a formula defined inductively as follows: (i) if G consists of a single leaf node labelled by a formula

P, then G represents P; (ii) if G has a root node labelled X with edges labelled $A_1 \dots A_n$ leading to subgraphs $G_1 \dots G_n$, and each G_i represents a formula P_i , then G represents the formula $\bigvee_{1 \leq i \leq n} ((X = A_i) \wedge P_i)$. Like ROBDDs, MDGs are also reduced and ordered.

The temporal logic used in MDG model checker is L_{MDG} . The atomic formulas of L_{MDG} are the constants True or False, and equation $t_1=t_2$, where t_1 is an ASM-variable, t_2 is an ASM-variable, a constant, an ordinary variable or a function of ordinary variables. An ASM-variable is a variable appearing in the description of an abstract state machine (ASM). Ordinary variables appear in the specifications of properties, which are used to remember the values of some ASM-variables at the current state. If p, q are L_{MDG} formulas, then so are $!p$, $p \& q$, $p|q$, $p \rightarrow q$, LET ($v=t$) IN p , Xp , Ap , AGp , AFp , $A(p \cup q)$, $AG(p \rightarrow F(q))$, $AG(p \rightarrow (q \cup r))$. In the formula LET ($v=t$) IN p , v is an ordinary variable and t is an ASM-variable. The semantics are defined on an abstract computation tree [3, 4]. A property in L_{MDG} holds on an ASM if and only if the property is true for all the paths starting from the initial states in the abstract computation tree.

3 Property Based Model Reductions

In the transition system of a model M, let $Y=(y_1, \dots, y_n)$ be the state variables, $Y'=(y_1', \dots, y_n')$ the corresponding next state variables, and $X=(x_1, \dots, x_m)$ be the input variables. Let f_i be the next state function, then $y_i'=f_i(Y, X)$. The transition relation of the state variable y_i is $T_i(Y, X, y_i') \Leftrightarrow (y_i'=f_i(Y, X))$.

Definition 1. Let $ddv(y_i)$ be the set of direct determining variables of y_i . It includes all variables $v \in Y \cup X$ such that $f_i|_{v=a} \neq f_i|_{v=b}$ for some $a \neq b$, where $f_i|_{v=x}$ is the cofactor of f_i for $v=x$. Let $dv(y_i)$ denote the set of determining variables of y_i , which is defined recursively as follows: $dv(y_i)=ddv(y_i) \cup \{dv(y_j) \mid y_j \in ddv(y_i) \setminus y_i\}$. Then, let $ddsv(y_i)$ be the set of direct determining state variables of y_i , $ddsv(y_i)=ddv(y_i) \setminus X$, and $dsv(y_i)$ denote the set of determining state variables of y_i , $dsv(y_i)=dv(y_i) \setminus X$. The variables that are not in $dv(y_i)$ are called *don't care variables* of y_i . For a set of state variables $SetV=(y_1, \dots, y_k)$, $ddv(SetV)=ddv(y_1) \cup \dots \cup ddv(y_k)$, $dv(SetV)=dv(y_1) \cup \dots \cup dv(y_k)$, $ddsv(SetV)=ddv(SetV) \setminus X$, $dsv(SetV)=dv(SetV) \setminus X$.

Definition 2. Let P be the property to be verified, and $Y_P=(y_1, \dots, y_k)$ be the state variables appearing in P. The set of determining variables of the P is $dv(Y_P)=dv(y_1) \cup \dots \cup dv(y_k)$, and the set of determining state variables is $dsv(Y_P)=dv(Y_P) \setminus X$. Let VP be the set of property dependent state variables of P, and $VP=Y_P \cup dsv(Y_P)$.

Definition 3. Let S_1 and S_2 be two sets of state variables and $S_1 \cap S_2 = \emptyset$. If $dv(S_1) \cap dv(S_2) = \emptyset$, which means that S_1 and S_2 have no common determining variables, then S_1 and S_2 are *noncorrelated*.

The transition relation of M is a conjunction of the individual transition relations of the state variables. Constructing the total transition relation often consumes large memory and limits the applications of symbolic model checking. If the prop-

erty to be verified is only affected by a part of the system behaviour, we can only use the corresponding subset of the transition relations. The size of the state space representation can thus be reduced. In our case, we reduce all state variables that cannot directly or indirectly affect the state variables in P to primary inputs. Consequently the property holds on the original system if and only if it holds on the system constructed using state variables in VP only. But the resulting system may still be too large. We can use the transition relations of a subset of the variables in VP to construct the abstract model M' , however, the ACTL property P is only weakly preserved now. This means that if P holds on M' , it also holds on M , but the opposite may not be true. The difficulty lies in selecting the best subset of VP to verify the property. In the following section we will introduce an heuristic iterative reduction algorithm.

4 An Iterative Reduction Algorithm for Property Checking

We can view the dependency of property P on the determining variables as a directed graph, called *property dependency graph (PDG)*. The root of the graph contain Y_P , the state variables in P . A node in the graph contains one determining variable of Y_P which can be a state variable or an input. If there is an edge $v \rightarrow w$ in PDG, it means that w is a direct determining variable of v . Starting from the state variables in P , PDG can be easily constructed and it might be a cyclic graph.

We explain the basic idea behind our algorithms. For instance, in the property dependency graph given in Figure 1, y_1 is directly determined by y_2 and y_3 , and the two subgraphs $G_1 \cap G_2 = \emptyset$, which means that y_2 and y_3 have no common determining variables. Thus the values of y_2 and y_3 have no correlation. When we consider the influence of y_2 on y_1 , we can leave y_3 as a primary input, or vice versa. For verifying the property $AG(y_1=1)$, we use $G_1 \cup \{y_1\}$ to construct the reduced model and change the state variables in G_2 to primary inputs. If the property holds on the reduced model, the procedure terminates, otherwise we use $G_2 \cup \{y_1\}$ to construct the reduced model and change the state variables in G_1 to primary inputs. If the property still fails, then we use all the state variables in $G_1 \cup G_2 \cup \{y_1\}$ to verify the property. This procedure can be further refined, since using all state variables in a subgraph may not be necessary and may still produce a large state space. Beginning from the root of PDG, we search through the graph level by level. The root is level 0, the nodes that can be reached by n edges from the root are in level n . Since there may be cycles in PDG, we mark the variables as visited which were used to construct reduced models. When a visited variable is reached again, it is removed from the set of the newly reached variables, then the further search beginning from this variable is stopped. The initial reduced model is constructed by Y_P , the state variables in P and these variables are marked visited. If this model does not satisfy P , we get all the direct determining variables of Y_P by searching the nodes of level 1 in PDG. We partition the newly reached state variables into non-correlated sets, and then add the smallest set (S) of them to the current set of state variables which was previously used to construct a reduced model. If the property fails on this model, we again search through the graph beginning from S , and repeat the above

procedure. We thus iteratively add state variables to construct a more and more complete model. It can be viewed as a depth-first search of a modified property dependency graph in which a node represents a noncorrelated set of state variables.

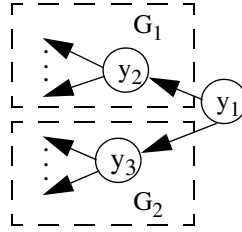


Figure 1. Example of a property dependency graph

The iterative reduction algorithm *reduction_verify*(M,P) is shown in Figure 2. Here M is the design description file, and P is the property to be verified. Y_P contains the state variables appearing in P. Y' is the set of state variables that is used to compute the abstract model. VP is the set of property dependent state variables of P.

```

reduction_verify(M, P)
begin
   $Y_P$  is the set of the state variables in P;
  PDG= construct_PDG( $Y_P$ , M);
  VP is the set of the state variables in PDG;
   $Y' := Y_P$ ;
  Mark the state variables in  $Y'$  as visited;
  result := verify(M, P, VP, PDG,  $Y'$ ,  $Y'$ );
  if result == success
    print('Property checking succeeded');
  else print('Property checking failed');
end;

verify(M, P, VP, PDG,  $Y'$ , lastset)
begin
   $M' := \text{reduce\_model}(M, Y')$ ;
  result := modelcheck ( $M'$ , P);
  if result == success
    return success;
  else if  $Y' == VP$ 
    return failure;
  else

```



```

begin
  ddsv := find_ddsv(lastset, PDG);
  ddsv1 := remove_visited_variables(dds1);
  listofsets := partition_set(dds1, PDG);
  listofsortedsets := sort_list(listofsets);
  dsv := find_dsv(lastset, PDG);
  sortedlist := append_to_tail(dsv, listofsortedsets);
  while sortedlist is not empty
  begin
    newset := head of sortedlist;
    remove newset from sortedlist;
    Mark the state variables in the newset as visited;
    newY' := Y'  $\cup$  newset;
    verify(M, P, VP, PDG, newY', newset);
  end
end
end

```

Figure 2. The iterative reduction algorithm

The subprocedure *construct_PDG*(Y_p , M) constructs the property dependency graph. *find_ddsv*(S , PDG) finds all direct determining state variables of the set S by searching through PDG . *find_dsv*(S , PDG) finds all determining state variables of the set S . *remove_visited_variables*($dds1$) removes the visited state variables from the set of the newly reached state variables. *reduce_model*(M , Y') reduces other state variables except Y' to primary inputs. *partition_set*($Vars$, PDG) partitions the variables in $Vars$ into noncorrelated sets. *sort_list* sorts the sets in increasing order of their sizes. *verify* accomplishes the iterative reduction and model checking.

We use the example in Figure 3 to illustrate our algorithm. y_1, \dots, y_7 are state variables, x_1, x_2 are primary inputs, the property to be verified is $AG(y_7=1)$. During the following iterative reduction, if the property checking succeeds at any iteration of the reduction algorithm, the verification terminates. Initially $Y'=(y_7)$. If the property fails on this model, $dds1(y_7)$ is divided into 2 noncorrelated sets S_1 and S_2 . The sorted list is $L_1=[S_1, S_2, dsv(y_7)]$, where $dsv(y_7)=(y_1, y_2, y_3, y_4, y_5, y_6)$. The smaller set (S_1) is added to Y' , i.e., $Y'=S_1 \cup (y_7)$. If the property does not hold on the abstract model, S_2 is selected, and we get $Y'=S_2 \cup (y_7)$. If the property fails on this model, $dds1(S_2)$ is divided into two noncorrelated sets S_3 and S_4 . The sorted list becomes $L_2=[S_3, S_4, dsv(S_2)]$, where $dsv(S_2)=(y_1, y_2, y_3)$, i.e., $dsv(S_2)=S_3 \cup S_4$. S_3 is added to Y' to get $Y'=S_3 \cup S_2 \cup (y_7)$. If the property fails again, S_4 is selected, and $Y'=S_4 \cup S_2 \cup (y_7)$. If the property still fails, $dsv(S_2)$ is selected and $Y'=S_4 \cup S_3 \cup S_2 \cup (y_7)$. If the property fails again, since L_2 is empty, we go back to L_1 and select $dsv(y_7)$ to get $Y'=S_4 \cup S_3 \cup S_2 \cup S_1 \cup (y_7)$. Now all the determining state

variables of the property have been used, the verification is final, and the property is strongly preserved.

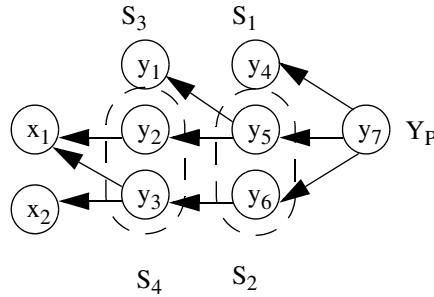


Figure 3. A property dependency graph

Since there are no redundant nodes and no redundant subgraphs in an MDG, the state variables appearing in the MDG of the transition relation of the state variable y are the direct determining variables of y . By scanning MDGs, we can easily construct the property dependency graph.

Properties are expressed by L_{MDG} formulas for the MDG model checker [3, 4]. When a property P is parsed by MDG, an additional circuit M_P representing P is so constructed that to verify P on the design model is to verify a simplified property on the composite machine of the design model and M_P . The simplified property is $A((flag1 = 1)U(flag2 = 1))$ for $A((Next_let_formula1) U (Next_let_formula2))$, $AG(flag=1)$ for $AG(Next_let_formula)$, $AF(flag=1)$ for $AF(Next_let_formula)$, where $flag$, $flag1$ and $flag2$ are boolean state variables of M_P which indicate the truths of the $Next_let_formulas$ one clock cycle earlier. Let $M_P = (X, Y, F_I, F_T)$. The input variables X of M_P are the ASM-variables of the design model, which appear in the original property P . Let n be the maximum nesting number of X operators in P . The state variables Y and the transition relation F_T are constructed to remember the values of input variables of M_P or the results of the comparison of variables in the past n (or less than n) clock cycles. The initial values F_I are defined so as not to affect the result of verifying P on the original design. After M_P is constructed, the simplified property is verified on the composite machine consisting of M and M_P . The initial reduced model here is not constructed by the set of state variables appearing in the simplified property, but by the set of the direct determining state variables of $flag$ and the state variables in M_P which directly represent the original property.

5 A Case Study by Using MDG, FormalCheck, and SMV

Consider the example [5] in Figure 4. The structure of the circuit is quite common in data processing circuits. The appropriate context (set of registers, memory data,

etc.) is selected based on the control signals (address of the memory, etc.), processing is carried out on the selected context, and then the modified context is stored in the same memory element. It is also quite common in telecommunication circuits in which channel or link numbers select the corresponding registers to be updated. This structure can be easily enlarged by adding more registers and increasing the size of the registers.

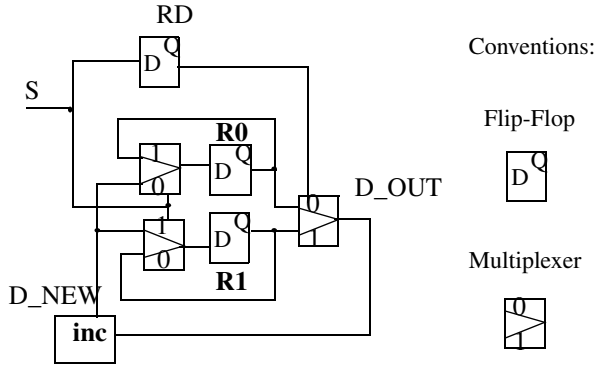


Figure 4. A data processing circuit M

In Subsection 5.1, all the signals in the circuit are defined in boolean level. We use three verification tools MDG, FormalCheck and SMV to verify the models with different number and size of registers. The experimental results are then discussed. Since in MDG there are abstract variables and uninterpreted functions, we give an abstract description of the circuit in Subsection 5.2 and verify it using MDG. The experimental results are compared. All of the experiments were carried out on a workstation Sun Ultra 10 with 333 MHz and 1GB of memory. In the following tables, ‘-’ means that the verification did not terminate.

5.1 Boolean Level Descriptions of the Model

We define all the signals in the circuit shown in Figure 4 to be a boolean variable. The registers are defined to have a certain number of bits, and each bit is represented by one boolean signal. Property P1 to be verified states that if S is 0, RD is 0, and the value of register R0 is 0 in the current clock cycle, then the value of R0 will be 1 in the next clock cycle. Property P2 to be verified states that if S is 0, RD is 0 and the value of R0[0] is 0 in the current clock cycle, then the value of R0[0] will be 1 in the next clock cycle. We verified these two properties on the models with different register numbers and sizes.

Table 1 shows the experimental results by using MDG. From Table 1, we can see that without our reduction algorithm, MDG can only verify the models having two registers with widths less than 20 bits. The reduction algorithm has significantly increased the useful domain of MDG. When the number of registers is increased to 12 and the width is increased to 28 bits, P1 and P2 can still be verified by MDG

with the reduction algorithm. P1 illustrates one behaviour of R0, which only refers to the boolean signals of R0, but does not refer to the other registers. Our algorithm automatically reduces all boolean signals of the other registers. From this table, we can see that for the models with registers of 28 bits, no matter how many registers are added, the state variables used to verify P1 are always 31 which include R0[0],..., R27[0], and three additional state variables for the property representation. P2 only refers to R0[0], and our algorithm reduces the other bits of R0 and all the other registers. Table 1 shows that only 4 state variables are used to verify P2 no matter how many and how large the registers are in the model.

Table 1: Experimental results with MDG

Property	Register No. & Width	No reduction				Reduction			
		State Vars	Nodes	Time (Sec)	Mem (MB)	State Vars	Nodes	Time (Sec)	Mem (MB)
P1	2 & 8	20	1554	2.42	2.05	11	1249	2.41	1.72
	2 & 12	28	2636	3.17	2.87	15	2109	3.73	2.41
	2 & 16	36	4083	4.73	3.94	19	3131	5.60	3.24
	2 & 20	-	-	-	-	23	5635	10.26	4.77
	2 & 28	-	-	-	-	31	9390	19.56	8.54
	12 & 28	-	-	-	-	31	42247	3863.6	517.8
P2	2 & 8	20	1199	1.77	1.90	4	506	1.86	1.36
	2 & 12	28	2003	2.58	2.59	4	686	2.32	1.80
	2 & 16	36	3057	3.66	2.46	4	866	3.22	2.30
	2 & 20	-	-	-	-	4	1092	5.27	3.19
	2 & 28	-	-	-	-	4	1468	11.51	5.81
	12 & 28	-	-	-	-	4	23447	4006.6	507.3

We use the model as an example which has two registers R0 and R1 with 28 bits. By using MDG, the model is written in MDG-HDL which does not have the means to handle arrays. This makes the description quite long and is not included here. We use Property P2 to illustrate how the reduction algorithm works. Property P2 is expressed by the following L_{MDG} formula: $AG((s = 0 \ \& \ rd = 0 \ \& \ r0_0=0) \rightarrow (X(r0_0=1)))$; the additional ASM extracted from P2 is shown in Figure 5.

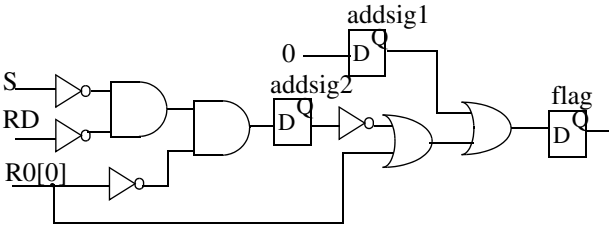


Figure 5. The additional ASM M_{p2} for P2

In Figure 5, flag, addsig1 and addsig2 are state variables. The initial values of flag and addsig1 are 1, which guarantees that flag=1 during the first 2 clock cycles. The verification of P2 on the original model is transferred to the verification of P2': $AG(flag = 1)$ on the composite ASM consisting of M and M_{P2} .

The property dependency graph of P2' is shown in Figure 6, in which node S is the primary input and the other nodes are the state variables. At the first iteration of the reduction algorithm, the set {flag, addsig1, addsig2, R0[0]} is used to construct the reduced model, and the simplified property $AG(flag=1)$ is verified. Thus only 4 state variables are used to verify P2 which is shown in Table 1. In the worst case if the design has some errors and P2 should fail, i.e., the initial reduced model constructed by {flag, addsig1, addsig2, R0[0]} cannot satisfy the property $AG(flag=1)$, then the set {RD, R1[0]} is added to construct the model for the second iteration. Now all the state variables in the property dependency graph are used, and the verification is final. To verify P2, only 6 state variables are needed in the worst case.

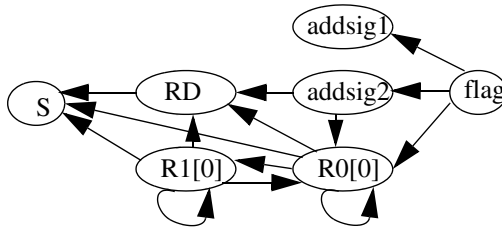


Figure 6. The property dependency graph of P2'

Now we use FormalCheck and SMV to verify the models of the circuit shown in Figure 4. We again use the model with 2 registers of 28 bits. The model in synthesizable Verilog for FormalCheck is as follows:

```

module main(S, RST, CLK, D_OUT);
  input S, RST, CLK;
  output [27:0] D_OUT;
  reg [27:0] R0, R1;
  reg RD;
  wire [27:0] D_NEW, D_OUT;
  assign D_OUT=(RD==1'b0)? R0 : R1;
  assign D_NEW= D_OUT+{{6{4'b0000}},4'b0001};

  always @(posedge CLK)
  begin
    if (RST==1'b0)
    begin {R0,R1}<={8{7'b0000000}}; end
    else begin
      RD<=S;
      case (S)
        1'b0: R0<=D_NEW;
        1'b1: R1<=D_NEW;

```

```

        endcase
    end
end
endmodule

```

Property P1 expressed in FormalCheck becomes: After: (CLK==rising && RST!=0 && S==0 && RD==0 && R0==0) Always: R0==1 Unless: CLK==rising. Property P2 becomes: After: (CLK==rising && RST!=0 && S==0 && RD==0 && R0[0]==0) Always: R0[0]==1 Unless: CLK==rising.

In SMV the above model and the properties P1 and P2 were rewritten in Synchronous Verilog (SV), which requires some modifications to the original Verilog code. P1 expressed in SV is as follows:

```

always
begin
    if (rst==1 & S==0 & RD==0 & R0==0)
        begin wait(1); assert R0_update: R0==1; end
    end
end

```

The experimental results are shown in Table 2. For SMV, Time indicates user time, while for FormalCheck and MDG, it is real time including loading the Verilog or MDG-HDL description file, compilation and model checking. The reduction algorithm selected in FormalCheck is Iterated with Empty reduction seed, and the run option is Symbolic (BDD). The run option of SMV is the one which uses the heuristic variable ordering, computes the number of reachable states and restricts model checking to reachable states. For MDG the iterative reduction algorithm is selected.

Table 2: Experimental results with FormalCheck, SMV and MDG

Property	Register No. & Width	FormalCheck			SMV			MDG		
		State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)	State Vars	Time (Sec)	Mem (MB)
P1	2 & 16	26	26	3.39	34	0.29	8.40	19	5.6	3.24
	2 & 20	30	26	3.48	42	0.38	8.42	23	10.26	4.77
	2 & 28	38	37	3.73	58	0.63	8.42	31	19.56	8.54
	4 & 28	39	67	5.04	115	5.79	8.57	31	72.04	29.85
	8 & 28	40	131	9.24	228	7.28	9.52	31	759.8	165.7
	10 & 28	41	213	12.84	285	10.87	10	31	1820	315.3
	12 & 28	41	301	16.30	-	-	-	31	3863	517.8
P2	2 & 16	38	1772	68.12	4	0.06	8.24	4	3.22	2.30
	2 & 20	49	32760	961.2	4	0.08	8.25	4	5.27	3.19
	2 & 28	-	-	-	4	0.08	8.25	4	11.51	5.81
	4 & 28	-	-	-	7	0.18	8.37	4	66.32	26.04
	8 & 28	-	-	-	12	0.49	9.03	4	780.1	158.8
	10 & 28	-	-	-	15	0.81	9.42	4	1886	306.6
	12 & 28	-	-	-	17	1.09	9.84	4	4006	507.3

Many different factors influence the experimental results, e.g., these tools use different variable ordering, different partitioning of the transition relations and different reduction methods. MDG graphs are much larger than the others for concrete representations, because the MDG structure and algorithms are more complicated than those of ROBDD. But the columns of state variables illustrate that our reduction algorithm can reduce the models appropriately according to the properties to be verified. For Property P1 our reduction algorithm reduced all the registers other than R0. From Table 2 we can see that for the models with 2, 4, 8, 10, 12 registers of 28 bits, the state variables used are always 31 including R0[0],..., R0[27] and 3 state variables in the additional ASM representing P1. However, SMV used all registers to verify P1, and when the model was enlarged to have 12 registers of 28 bits, SMV could not complete the verification. FormalCheck used more state variables when the number of registers in the model was increased. After verifying P1 on the model with 2 registers of 28 bits, we opened the reduction manager window to see that R0[0],...,R0[27], R1[0],...,R1[3] were used, yet R1[0],...,R1[3] could have been reduced. For Property P2 our reduction algorithm used only 4 state variables (R0[0], flag, addsig1, addsig2) and automatically reduced the other bits of R0 and all the other registers no matter how many and how large the registers are. SMV used the least significant bit variables of all the registers to verify P2, e.g., for the model with eight registers of 28 bits, R0[0],...,R7[0] are used by checking the “cone” window in SMV. Table 2 shows that the number of state variables used in SMV is growing when more registers are added in the models. FormalCheck could not reduce anything when verifying P2. After verifying P2 on the models with 2 registers of 16 or 20 bits, we opened the reduction manager window and saw that all state variables were used. When the models became even larger, FormalCheck could not complete the verification of P2.

5.2 An Abstract Description of the Model

For the circuit shown in Figure 4, we are only concerned about the data in a selected register being correctly updated and stored in the appropriate register. We can define the registers as words of size n using abstract sorts, e.g. an abstract sort `wordn`. The high-level words are of arbitrary size, which makes the description generic, and the verification is thus applicable to registers of any word size. For the data processing unit, here an incrementer, we can use an uninterpreted function `finc`. MDG allows abstract variables and uninterpreted functions and `wordn` and `finc` are defined as of abstract sort in the algebraic file:

```
abs_sort(wordn).
function(finc, [wordn],wordn).
```

The abstract model M in MDG-HDL is as follows:

```
% Variables definition
signal(s,bool).
signal(d,wordn).
signal(n_r0, wordn).
signal(r0, wordn).
```

```

signal(n_r1, wordn).
signal(r1, wordn).
signal(n_rd, bool).
signal(rd, bool).
signal(finc_out, wordn).
signal(finc_in, wordn).
% Pairs of a state variable and its next state variable
st_nxst(r0, n_r0).
st_nxst(r1, n_r1).
st_nxst(rd, n_rd).
% Circuit definition
component(fork_s, fork(input(s), output(n_rd))).
component(mux1,mux(sel(s),inputs([(0,finc_out),(1,r0)]),output(n_r0))).
component(mux2,mux(sel(s),inputs([(1,finc_out),(0,r1)]),output(n_r1))).
component(mux3,mux(sel(rd),inputs([(0,r0),(1,r1)]),output(finc_in))).
component(r0,reg(input(n_r0),output(r0))).
component(r1,reg(input(n_r1),output(r1))).
component(rd,reg(input(n_rd),output(rd))).
component(finc,transform(inputs(finc_in),function(finc),output(finc_out))).
outputs([]).
output_partition([]).
next_state_partition([[n_r0]],[n_r1],[rd]]).
par_strategy(auto,auto).

```

The property to be verified on the abstract model specifies that if S is 0 and RD is 0, and the value of $R0$ is v in current clock cycle, then $R0$ will be the value of $finc(v)$ in the next clock cycle. The property expressed in L_{MDG} is as follows:

P3: $AG(LET(v=r0) \text{ IN } ((s = 0 \ \& \ rd = 0) \rightarrow (AX \ (r0 = finc(v))))$;

The circuit in Figure 7 represents the additional ASM for P3. The verification of P3 on the original model is transferred to the verification of P3': $AG(flag = 1)$ on the composite ASM consisting of M and M_{P3} .

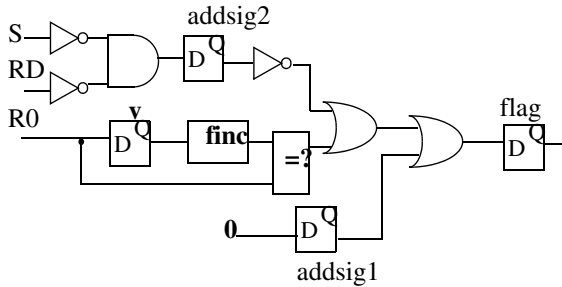


Figure 7. The additional ASM M_{P3} for P3

The individual transition relations of the composite machine are represented by Directed Formulas [1, 2], which translate immediately to the MDG graph representations.

$$\begin{aligned}
T_{\text{flag}}: & ((\text{addsig1}=0) \wedge (\text{addsig2}=1) \wedge (\text{abscomp}(\text{finc}(v), R0)=0) \wedge (\text{flag}'=0)) \vee \\
& (((\text{addsig1}=1) \vee (\text{addsig2}=0) \vee (\text{abscomp}(\text{finc}(v), R0)=1)) \wedge (\text{flag}'=1)) \\
T_{\text{sig1}}: & (\text{addsig1}'=0) \\
T_{\text{Sig2}}: & (((RD=1) \vee (S=1)) \wedge (\text{addsig2}'=0)) \vee ((RD=0) \wedge (S=0) \wedge (\text{addsig2}'=1)) \\
T_v: & (v'=R0) \\
T_{RD}: & (RD'=S) \\
T_{R0}: & ((S=1) \wedge (R0'=R0)) \vee ((S=0) \wedge (RD=0) \wedge (R0'=\text{finc}(R0))) \vee \\
& ((S=0) \wedge (RD=1) \wedge (R0'=\text{finc}(R1))) \\
T_{R1}: & ((S=0) \wedge (R1'=R1)) \vee ((S=1) \wedge (RD=1) \wedge (R1'=\text{finc}(R1))) \vee \\
& ((S=1) \wedge (RD=0) \wedge (R1'=\text{finc}(R0)))
\end{aligned}$$

The property is now verified in 0.7 second in 1.04M of memory, containing 201 MDG nodes. The state variables (RD, R1) are automatically reduced to primary inputs, and only the state variables (flag, addsig1, addsig2, v, R0) are used. However without reduction, P3 is successfully verified in 1.5 seconds in 1.17M of memory, containing 230 MDG nodes. Data abstraction makes the verification very fast and is applicable to any datapath width. By combining abstract data representation with efficient model reductions the state space explosion problem can be considerably diminished.

6 Conclusions

We have introduced an heuristic model reduction algorithm. Since there is no pre-image operation in MDG due to the presence of abstract state variables, all backward reduction algorithms cannot be used in MDG. Our method is suitable for MDG, however, it can be used in other tools as well. Moreover this algorithm is completely automatic without any user-guided information. For a given property it produces and analyzes the property dependency graph, and iteratively adds noncorrelated sets of state variables to construct a series of reduced models. We studied a common circuit structure that appears in telecommunication and data processing circuits. We used MDG, SMV and FormalCheck to verify this circuit, the experimental results were compared. This reduction algorithm has significantly increased the useful domain of MDG, and can achieve more efficient reduction based on the property to be verified. Although this example is simple and trivial, it is sufficient to illustrate that our algorithm can do appropriate reduction in a large domain of real circuits which have the same structure as the example. We verified this on a scheduler circuit from an ATM switch containing 118 state variables (69 abstract state variables and 49 concrete state variables) and 120 abstract functions. Without the reduction algorithm, verifications could not complete, while using our reduction algorithm, one property was verified in 15 minutes with 7 state variables used, and another property was verified in 4.7 minutes with 5 state variables used. Right now there is no

efficient ordering algorithms in MDG. Since MDG has abstract variables and cross-terms, this makes the node ordering more complicated than ROBDD. We are now developing good static and dynamic ordering algorithms in MDG.

Acknowledgment

The work was partially supported by an NSERC-Nortel CRD Grant, and the experiments were carried out on workstations on loan from the Canadian Microelectronics Corp.

References

- [1] F.Corella, Z.Zhou, X.Song, M.Langevin, E.Cerny. Multiway Decision Graphs for Automated Hardware Verification. In Formal Methods in System Design, 10(1),1997.
- [2] Z.Zhou, X.Song, F.Corella, E.Cerny, M.Langevin. Description and Verification of RTL Design Using Multiway Decision Graphs.Technical Report RC19822,IB T.J.Watson Research Center, November 1994.
- [3] Y. Xu. Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs. Ph.D thesis, University of Montreal, 1999.
- [4] Y. Xu, E.Cerny, X.Song, F.Corella, O. Mohamed. Model Checking for a First-Order Temporal Logic using Multiway Decision Graphs. In Proceedings of Conference on Computer Aided Verification (CAV 98), July 1998.
- [5] Y. Xu, E.Cerny, A.Silburt, A.Coady, Y.Liu, P.Pownall. Practical Application of Formal Verification Techniques on a Frame Mux/Demux Chip from Nortel Semiconductors. Charme 1999.
- [6] E. Allen Emerson. Temporal and Modal Logic, 16th chapter of Hand book of Theoretical Computer Science, edited by J.van Leeuwen. Elsevier Science Publishers B.V 1990.
- [7] R.E.Bryant. Graph-based Algorithms for Boolean Function Manipulation. In IEEE Transactions on Computers, 35(8), August 1986.
- [8] J.R.Brurch, E.M.Clarke, D.E.Long, K.L.McMillan, D.L.Dill. Symbolic Model Checking for Sequential Circuit Verification. In IEEE Transactions on Computer-Aided Design, 13(4), April 1994.
- [9] K.L.McMillan. Symbolic model checking - an approach to the state explosion problem. Ph.D thesis, SCS, Carnegie Mellon University, 1992.
- [10] J.R.Brurch, E.M.Clarke, K.L.McMillan. Symbolic Model Checking: 10^{20} States and Beyond. In proceeding of LICS 1990.
- [11] E.Clarke, O.Grumberg, and D.Long. Verification tools for Finite-State Concurrent Systems, Lecture Notes in Computer Science 803, 1994.
- [12] E.Clarke, O.Grumberg, and D.Long. Model Checking and Abstraction. In ACM-TOPLAS, Vol 16, No.5, September 1994.
- [13] E.M.Clarke, T.Filkorn, S.Jha. Exploiting Symmetry in Temporal Logic Model Checking. In Formal Methods in System Design, Vol 9, 1996.
- [14] C. Norris Ip, David Dill. Better Verification Through Symmetry. In Formal Methods in System Designs, Vol 9, August 1996.

- [15] H.J. Touati, R.K.Brayton, R. Kurshan. Testing Language Containment for w-Automata Using BDDs. in *Information and Computation* 118, 1995.
- [16] R.P.Kurshan. Reducibility in Analysis of Coordination. In *Lecture Notes in Control and Information Sciences*. Vol 103, 1987.
- [17] R.P.Kurshan. Analysis of Discrete Event Coordination. In *Lecture Notes in Compute Science*. Vol 430, 1990.
- [18] M. Sela, F. Limor, I. Haifa. Input Elimination and Abstraction in Model Checking. *FMCAD98*, Nov 3-6, 1998.
- [19] G.Cabodi, P.Camurati, S.Quer. Improved Reachability Analysis of Large Finite State Machines. *ICCAD 96*, Nov 10-14, 1996.
- [20] C. Norris Ip, David Dill. State Reduction Using Reversible Rules. In *33rd Design Automation Conference*, Las Vegas, June 1996.
- [21] H. Higuchi, Y. Matsunaga. A Fast State Reduction Algorithm for Incompletely Specified Finite State Machines. In *33rd Design Automation Conference*, Las Vegas, June 1996.
- [22] E.A. Emerson, A.P. Sistla. Symmetry and Model Checking. In *CAV93*, June 1993.
- [23] Dennis Rene Dams. Abstract Interpretation and Partition Refinement for Model Checking. Ph.D thesis, Eindhoven University of Technology, 1996.
- [24] Dennis Dams, Orna Grumberg, Rob Gerth. Generation of Reduced models for Checking Fragments of CTL. In *CAV93*, June 1993.
- [25] David Lee, Mihalik Yannakakis. Online Minimization of Transition Systems. *24th Annual ACM STOC* 1992.
- [26] K. Fisler, Moshe Y. Vardi. Bisimulation Minimization in an Automata-Theoretic Verification Framework. *FMCAD98*, Nov 3-6, 1999.

Modeling and Parameters Synthesis for an Air Traffic Management System

Adilson Luiz Bonifácio and Arnaldo Vieira Moura

Computing Institute, University of Campinas
P.O. 6176, Campinas, Brazil, 13081-970
{adilson,arnaldo}@dcc.unicamp.br

Abstract The aim of this work is to apply formal specification techniques to model real-time distributed systems arising from real-world applications. The target system is an Air Traffic Management System (ATM), which uses the Traffic alert and Collision Avoidance System (TCAS) protocol. The formal models developed here are based on the notion of hybrid automata. Semi-automatic tools are used in the verification of the models, and some important system parameters are synthesized using parametric analysis. All results were obtained on a 350MHz desktop PC, with 320MB of main memory.

1 Introduction

The use of formal specifications and verification techniques in real-world system development processes has become more and more important, especially when such processes focus on distributed systems that control critical applications, where an operational fault can cause irreparable damages. Among a wide variety of distributed systems, the ones involving reactive and real-time responses are the most difficult to model accurately. Certain kind of critical applications, known as *hybrid systems*, present a dual nature, in the sense that the system behavior is described by continuous dynamic profiles, regulated by the intervention of discrete events [1, 2, 3]. This introduces further complications and subtleties in the modeling and analysis of such systems.

Recently, the mathematical theory of hybrid automata has been gaining momentum as a good alternative to specify and verify hybrid systems [4, 5]. Essentially, hybrid automata are finite automata whose states include a dynamic specification of the system evolution, and whose state transitions model an abrupt change in the dynamic behavior of the system. Each component of the distributed system is modeled by an appropriate automaton. The overall system behavior is, then, captured by the corresponding product automaton [6]. The communication between the system components is regulated by the exchange of messages between the independent automata. Messages are exchanged during state transitions taking place in the individual automata. Synchronism between the individual automata is also attained by the use of a shared memory, to which all the component automata have access [7]. Hybrid automata are adequate to model systems composed of several distributed communicating components, each one with a simple dynamic behavior. In contrast, classic control theories deal with systems with a few distributed components whose dynamic features are more complex.

In this work, hybrid automata are used to model the complexities of a real-world distributed hybrid system, consisting of an Air Traffic Management System (ATM). The ATM system coordinates many different autonomous aircraft which compete for routes

in a section of airspace. The Traffic alert and Collision Avoidance System (TCAS) protocol is used to help manage the traffic in the vicinity of a cruising aircraft. The TCAS is a conflict detection and resolution algorithm that runs on an on-board embedded component. Its task is to monitor the traffic around the aircraft, detect possible threats and advise on how to resolve these conflicts. The operation of the ATM system requires an intense exchange of messages and a complex activity of coordination between all participating agents. Due to the complexity of the resulting model, (semi) automatic computational tools are used to effectively bear on the formal specifications. Most of the time, models for realistic systems result in an automaton of such complexity that the use of these automatic tools becomes essential. Here, the HyTech tool [10] was used.

The models explored in this work consider two aircraft flying in the same airspace, in opposite directions. This scenario captures only a fragment of the complexities of a real ATM system. Even so, the results obtained contribute a positive step towards the validation and the verification of the functionalities of the actual system. In all cases studied, a safe system operation is the main property to be verified. At the moment of this writing, the authors are unaware of other direct applications of hybrid automata techniques either to verify safety properties for ATM systems [11], or to synthesize values for operational parameters of such systems.

This work is organized as follows. Section 2 presents the notion of hybrid automata, with some simple examples. Section 3 describes the air traffic management system, its operation and some of its variants and special procedures. Section 4 discusses the models used to validate the system operation. The synthesis of values for important parameters is also described here. The last section offers some concluding remarks.

2 Hybrid Automata

A *hybrid automaton* [12, 13] is a system $A = (X, V, \text{flow}, \text{init}, \text{inv}, E, \text{jump}, \Sigma, \text{syn})$, where:

1. $X = \{x_1, \dots, x_n\}$ is a set of *variables*. The number n is the *dimension* of A .
2. V is a finite set of *operation modes*, or just *modes*.
3. For every $v \in V$, $\text{flow}(v)$, the *continuous activity condition* of v , is a predicate over the set of variables $X \cup \dot{X}$. Here, $\dot{X} = \{\dot{x}_1, \dots, \dot{x}_n\}$.
4. For every $v \in V$, $\text{inv}(v)$, the *invariant condition* for v , is a predicate over X .
5. The *initial conditions* are given by the *init* component. For every mode $v \in V$, $\text{init}(v)$ is a predicate over X .
6. The multi-set E is formed by pairs (v, v') , where $v, v' \in V$ are operation modes.
7. The *jump* component describes the *phase change conditions*. For every transition $e \in E$, $\text{jump}(e)$ is a predicate over the set $X \cup X'$. Here, $X' = \{x'_1, \dots, x'_n\}$. The primed variable x' is used to indicate the (new) value of x after a transition.
8. Σ is a set of *event labels*, or just *events*. The partial function syn maps E into Σ .

An example [14] of a single hybrid automaton is shown in Figure 1(a). This model captures the simple dynamics of a gas heater. The model uses a real variable x , which evolves deterministically in time, to measure the temperature of a container. The evolution of x is computed from the initial conditions, from differential equations present in the modes, and from the transitions between modes. The heater is turned off when the temperature reaches 3 degrees, and it is turned back on when the temperature falls

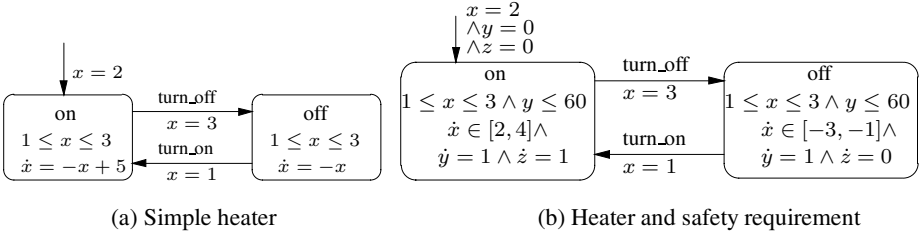


Figure 1. Hybrid automata: an example

to 1 degree. In the formal set up for this example $X = \{x\}$, and $V = \{on, off\}$. The continuous activity condition for mode *on* is given by $\dot{x} = -x + 5$, which describes an exponential rise of the temperature x . For mode *off* the continuous activity condition is $\dot{x} = -x$, describing an exponential fall in the temperature. For both operation modes, the invariant condition is $1 \leq x \leq 3$. The initial condition for mode *on* is $x = 2$, and for mode *off* it is *false*. The latter condition is not depicted in the figure. There are two transitions in Figure 1(a), namely (on, off) and (off, on) . The transition (on, off) has a phase change condition given by $x = 3 \wedge x' = x$, and the transition (off, on) has the phase change condition given by $x = 1 \wedge x' = x$. The trivial condition $x' = x$ is not shown in the figure. In this model, the value of the real variables do not change when a transition is taken. It is common to use “guards”, in the sense of [4], to represent the phase change conditions in the graphical representation of hybrid automata. A guard such as $(x_1 = x_2) \rightarrow (x_1 := 2x_2)$ gives the precondition $x_1 = x_2$ and the postcondition $x'_1 = 2x_2$, that are to be enforced during a phase change. These conditions are equivalent to the $x_1 = x_2 \wedge x'_1 = 2x_2 \wedge x'_2 = x_2$. The set of discrete events for this example is $\Sigma = \{turn_on, turn_off\}$. The function *syn* associates the event *turn_on* to the transition (off, on) , and associates the event *turn_off* to the transition (on, off) . The presence of these events will synchronize distinct distributed automata.

The model of a complex system comprises several individual automata that operate concurrently. The synchronism of the global system is obtained by: (i) the use of shared variables, and (ii) the simultaneity of phase transitions when they are labeled by the same discrete event. This mechanism is incorporated into the product automaton [8].

Let A_1 and A_2 are hybrid automata, where $X_1 \cap X_2 = \emptyset$. The *product automaton* of A_1 and A_2 is a system $A = (X_1 \cup X_2, V_1 \times V_2, flow, init, inv, E, jump, \Sigma_1 \cup \Sigma_2, syn)$, where: (i) $flow((v_1, v_2)) = flow_1(v_1) \wedge flow_2(v_2)$; (ii) $inv((v_1, v_2)) = inv_1(v_1) \wedge inv_2(v_2)$; (iii) $init((v_1, v_2)) = init_1(v_1) \wedge init_2(v_2)$. For the phase transitions, $e = ((v_1, v'_1), (v_2, v'_2)) \in E$ if and only if one of the following conditions hold:

1. $v_1 = v'_1, e_2 = (v_2, v'_2) \in E_2$ and $syn_2(e_2) \notin \Sigma_1$; $jump(e) = jump_2(e_2)$ and $syn(e) = syn_2(e_2)$.
2. $v_2 = v'_2, e_1 = (v_1, v'_1) \in E_1$ and $syn_1(e_1) \notin \Sigma_2$; $jump(e) = jump_1(e_1)$ and $syn(e) = syn_1(e_1)$.
3. $e_1 = (v_1, v'_1) \in E_1, e_2 = (v_2, v'_2) \in E_2, syn_1(e_1) = syn_2(e_2)$. In this case, $jump(e) = jump_1(e_1) \wedge jump_2(e_2)$ and $syn(e) = syn_1(e_1)$.

¹ For any variable x , the first derivative of x with respect to time is indicated by \dot{x} .

The product of several automata is obtained by accumulating the result of computing the product of each individual automaton, in turn [3, 4, 11].

An *atomic linear predicate* is an inequality between rational constants and linear combinations of variables with rational coefficients, such as $2x + 4y - 7z/2 \leq -10$. A *convex linear predicate* is a finite conjunction of atomic linear predicates, and a *linear predicate* is a finite disjunction of convex linear predicates. A hybrid automaton is *linear* when all the predicates, described in items 3–7 of its definition, are linear predicates.

A *convex region* of a hybrid automaton A of dimension n is a convex polyhedron in \mathbb{R}^n . A *region* is a finite union of convex regions. Given a predicate φ , the region determined by φ is denoted by $[\varphi]$, and is called the φ -region. A *configuration* of a hybrid automaton A is a pair $q = (v, a)$ consisting of an operation mode $v \in V$ and a vector $a = (a_1, \dots, a_n)$ in \mathbb{R}^n . The configuration (v, a) is *admissible* if $a \in [inv(v)]$. The configuration (v, a) is *initial* if $a \in [init(v)]$. In the example depicted on Figure 1(a), the configuration $(on, 0.5)$ is not admissible and the configuration $(on, 2)$ is initial.

Let $q = (v, a)$ and $q' = (v', a')$ be two configurations of A . The pair (q, q') is a *phase change* of A if $e = (v, v')$ is a transition in E and $(a, a') \in [jump(e)]$. The pair (q, q') is a *continuous activity* of A if $v = v'$, if there is a nonnegative $\delta \in \mathbb{R}$ (the duration of the continuous activity) and there is a differentiable function $\rho : [0, \delta] \rightarrow \mathbb{R}^n$ (the curve of the continuous activity), such that the following requirements are satisfied: (i) Endpoints: $\rho(0) = a$ and $\rho(\delta) = a'$; (ii) Admissibility: for all time instants $t \in [0, \delta]$, the configuration $(v, \rho(t))$ is admissible; (iii) Invariant: $\rho(t) \in [inv(v)]$, for all time instants $t \in [0, \delta]$; (iv) Continuous activity: if $\dot{\rho} : [0, \delta] \rightarrow \mathbb{R}^n$ is the first derivative of ρ , then, for all time instants $t \in [0, \delta]$, $(\rho(t), \dot{\rho}(t)) \in [flow(v)]$.

A *trajectory* of A is a sequence q_0, q_1, \dots, q_k , of admissible configurations, where each pair (q_j, q_{j+1}) of consecutive configurations is either a phase change or a continuous activity of A . Such a trajectory is said to *start* at q_0 . A configuration q' of A is *reachable from* a configuration q if q' is the last configuration of some trajectory of A starting at q . An *initial trajectory* of A is any of its trajectory that starts at an initial configuration. A configuration q' of A is *reachable* if it is reachable from an initial configuration of A . In Figure 1(a), all admissible configurations are reachable.

Given a region φ , $Post(\varphi)$ is the region formed by all those configurations q' for which there exists a configuration $q \in [\varphi]$ such that q' is reachable from q through a continuous activity or of a phase change of A . Starting with $\varphi_0 = \varphi$, the iteration of this process will compute the regions $\varphi_{k+1} = Post(\varphi_k)$, for $k = 0, 1, \dots$. If a region φ_k satisfies $\varphi_k = \varphi_{k+1}$, then the process converges and region φ_k contains all the configurations of A that are reachable via some trajectory that starts from a configuration in φ_0 . When the process converges, region φ_k is denoted by $Post^*(\varphi_0)$. A backward process can also be defined in a similar way, giving rise to a *Pre* operator.

The following theorem can be shown [11]: if A is a linear hybrid automaton and $[\varphi]$ is a region of A , then the calculation of $Post^*(\varphi)$ converges. This theorem supports the construction of computational tools for the analysis of linear hybrid automata.

A *safety requirement* is a set of predicates imposed on the system configurations. Usually, a safety requirement is described by the set of values that the system variables can attain. A configuration is *safe* if it satisfies all the safety requirements associated with the model; otherwise the configuration is *unsafe*. A model is *safe* if all its reachable configurations are safe; otherwise the model is *unsafe*. Given a safety requirement as a predicate φ over the configurations of A , the region $reach = Post^*([init])$ is computed, and the intersection $reach \cap [\neg\varphi]$ is obtained. If the resulting region is empty, the safety

requirements are satisfied and the system is safe; otherwise they are violated and the system is unsafe.

Returning to the heater example, depicted in Figure 1(a), a safety requirement for the heater could be: “in the first hour of operation, the heater should not be on by more than $2/3$ of the time”. Figure 1(b) depicts a relaxed version of that automaton, where the new clock variable y measures the total elapsed time and the new stopwatch variable z measures the time the heater stays in operation. Note that the conditions $1 \leq x \leq 3$ and $\dot{x} = -x + 5$ imply $\dot{x} \in [2, 4]$. Similarly for the *off* mode, resulting in a linear automaton. It can be demonstrated that the relaxed version is safe with respect to these requirements 1. Hence, the original model is also safe with respect to the same requirements.

System parameters are symbolic constants which assume fixed values 2. A parametric analysis determines value intervals for certain system parameters in such a way as to guarantee that no safety requirement is violated. This process, therefore, synthesizes maximum and minimum values for these parameters. In a hybrid automaton, a parameter α can be represented by a variable whose value never changes. This condition can be enforced by imposing the condition $\dot{\alpha} = 0$ in all operation modes and, further, by imposing the condition $\alpha' = \alpha$ over all transitions of the automaton. A value $a \in \mathbb{R}$ is *safe for a parameter* α if no unsafe configuration is reachable when the initial condition $\alpha = a$ is adjoined to the all operating modes of the automaton.

In general, the modeling of realistic systems results in a product automaton of such complexity that the verification phase can only be successfully done with the aid of (semi) automatic computational tools. The software HyTech 3 was developed as a tool to aid in the analysis and verification of linear hybrid automaton models. The model is described by an input file that comprises two parts. The first part describes the individual hybrid automata. The product automaton is computed automatically by the tool. The second part is a sequence of analysis commands. The HyTech tool can automatically verify that the relaxed heater model illustrated in Figure 1(b) satisfies all the posed safety requirements 1. Note that, when computing with symbolic parameters, the tool calculates the relationships that must hold among these parameters in order for the final region to be reached from the initial region. Thus, if the final region represents an unsafe condition, one can obtain safe intervals for the symbolic parameters by simply negating the computed output.

HyTech is one of the only tools that supports (semi) automatic analysis of linear hybrid automata models. It has been used in a variety of diverse situations and further examples can be found in 4. More recently, a successor to HyTech is being developed 5. UPPAAL 6 is another example of an automatic verification tool. It is based on timed-automata, a more restricted notion than that of linear hybrid automata. The former notion can only work with clocks, and does not support more complex differential equations to describe dynamic system profiles.

A number of other related works, dealing with models based on hybrid automata, can be found in 7.

3 Description of an Air Traffic Management System

Air Traffic Management Systems (ATM) are very complex and critical systems, that operate under tight conditions and are composed of several distributed components. This makes it hard for such systems to be formally verified. On the other hand, the need for a formal, rigorous, verification of all the ATM system safety requirements is unquestionable, since any safety fault can result in intolerably high damages, both in

terms of properties and lives. Several accidents are known to have occurred in aviation history, due to the unsafe operation of ATM systems [10].

In a typical flight, after the aircraft enter en route, the Air Traffic Control (ATC) monitors the flight by radio [11]. After an ATC acknowledges radar contact with the signaling plane, it starts to transmit values for heading and altitude, based on the aircraft present coordinates. Depending on the flight plan, one aircraft can switch many times from one en route controller to another, during the course of its flight. While all airline aircraft are controlled every step of the way, the same level of positive control does not always apply to all other aircraft. Some aircraft can, and often do, fly in uncontrolled airspace, outside the ATC range. In general, these uncontrolled airspaces are areas below the cruise lanes used by airline aircraft. General aviation aircraft are allowed to fly under visual flight rules, or VFR, when weather and visibility are good. In these conditions, they do not have to be in touch with air traffic control, unless they choose to operate in or out of an airport that has a control tower. Under VFR rules, pilots are responsible for maintaining adequate separation from other aircraft, and that is why these rules are sometimes called the “see and be seen” rules. Instrument flight rules, or IFR, on the other hand, are the rules under which general aviation aircraft must fly in bad weather and low visibility. In this case, pilots must be in contact with the ATC and must file a flight plan. They also must be “instrument certified”, meaning that they are proficient at navigating and flying their aircraft using cockpit instruments only, without the benefit of good visibility. Airline flights always operate under instrument flight rules, regardless of weather.

Recent technological advances made it possible to implement sophisticated functions, such as navigation and aircraft separation, in the embedded system components responsible for the control and surveillance of a sector of airspace [12]. This more advanced technology permits a reduction in the number of collision, while efficiently maintaining the throughput of the airspace with adequate levels of security and reliability. On the other hand, the use of more advanced technologies tightens further the system parameters and accrues the verification and safety problems, opening up the possibilities for the introduction of subtle errors. In these cases, a formal verification of the system safety is even more desirable.

Over the years, air traffic has continued to increase. The developments of modern air traffic control systems have made it possible to cope with this increase, while maintaining the necessary levels of flight safety. However, the risk of airborne collision remains. That is why the concept of an airborne collision avoidance system has been considered, giving rise to the initial development of the Traffic alert and Collision Avoidance System (TCAS) [13]. TCAS is a protocol used to monitor air traffic in the vicinity of flying aircraft. It provides the pilot with information about neighboring aircraft that may pose a collision threat and, also, it advises on how to resolve these conflicts. TCAS significantly improves flight safety. However, it can not entirely eliminate all collision risks. As in any predictive system, it might itself induce a risk of collision [14].

The TCAS system can enter in one of two levels of alertness [15]. In the first level, the system issues a Traffic Advisory (TA) signal to inform the pilot of a potential threat. When operating in this level of alertness, it does not provide any suggestions on how to resolve the situation. If the risk of collision increases, a Resolution Advisory (RA) signal is issued, and the system also suggests a maneuver that is likely to resolve the conflict. The TCAS system also allows for reversal commands and it may change the climb/descend advise during a conflict. This feature was added to the protocol in order to compensate for the nondeterminism in the pilot response. That is, the pilot may

choose not to follow the TCAS advises thereby rendering the original maneuver unsafe. The TCAS detects this situation and changes the RA, if necessary. This nondeterminism renders the necessity of a formal system verification even more necessary.

Aircraft separation standards vary according to circumstances. Above 29000 feet, when the aircraft are cruising at high speed, the standard is five miles of horizontal radar separation and 2000 feet of vertical separation. Below 29000 feet, the vertical separation is reduced to 1000 feet while the horizontal radar separation remains at five miles. When aircraft are moving at slower speeds, as when they are near an airport, the standard is three miles of horizontal radar separation and 1000 feet of vertical separation.

The TCAS protocol is designed to ensure collision avoidance between any two aircraft, with an approximation speed of up to 1200 knots and vertical rates as high as 10000 feet per minute. The controller action begins when the aircraft horizontal separation is 5 miles. The standard value for vertical climbs or descends is 10000 feet per minute. In emergencies it can be as high as 12000 feet per minute. The TCAS system monitors the vertical plane, and only a few texts consider the parallel separation between the aircraft. In the future, the TCAS system might produce RAs for both the horizontal and the vertical planes [14].

Aviation texts, usually, do not use the standard metric system. For example, the horizontal speed may be given in miles per hour and the vertical rate may be measured in feet per minute. Or, the vertical distance may be presented in feet and time may be given in minutes or hours, while the horizontal distance is given in miles. See Figure 1. In this work, however, all computed measures are converted to the standard metric system, so as to maintain a uniformity throughout [15]. The aim of this work is to validate

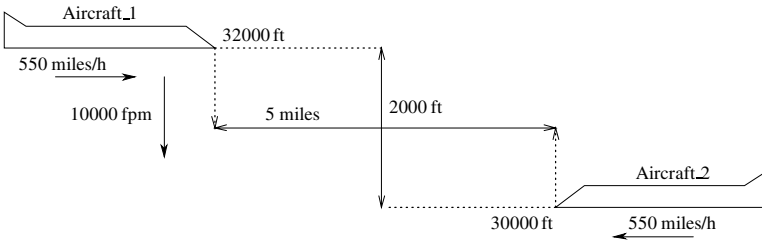


Figure 2. Two aircraft in the same airspace

the system operation, and to synthesize more appropriate and tighter values for some critical system parameters, in such a way to improve the system overall performance and to decrease the occurrence of unsafe situations. Another aspect to consider is the cooperation between the system parts.

Figure 2 depicts a situation with two aircraft in the same airspace. The changes of attitude for the aircraft are provoked by the controller that monitors the vertical and horizontal separations between the aircraft. Based on the position between the two aircraft, the on-board TCAS controller may command the leftmost aircraft to climb or to

² 1 foot \simeq 0.3048 meters, 1 mile \simeq 1829 meters and 1 knot \simeq 0.51 meters per second.

descend. It may also increase its descent rate, if the situation is critical. The rightmost aircraft is assumed to remain in a leveled cruise. The controller may, however, reduce its horizontal speed in order to avert collisions. In the scenario depicted in Figure 1, the leftmost aircraft is traveling at a height of 32000 feet and the rightmost one is traveling in the opposite direction, at a height of 30000 feet. The vertical distance between the aircraft is 2000 feet. The aircraft are supposed to travel at 550 miles per hour, when cruising. The speed of 490 miles per hour can also be applied in order to reduce the cruise speed. These rates and values are based on real data for Boeings 707 and 747.

The TA signal was suppressed from the models treated here and, when a conflict situation arises, a RA signal is issued directly. On the other hand, it is relatively easy to include more details of the TCAS protocols in the models, or to include more aircraft in the scenario. This would, of course, would lead to a more complex product automaton. The HyTech tool, however, was already operating close to its limits, running on a 350MHz desktop PC with 320MB of main memory. Certain internal limits of the tool, signaled by multiplication overflow errors, were also being reported. In order to accommodate a more complex and more complete model, it would be necessary to modify the source code of the tool and port it to a more powerful hardware platform.

More details about the TCAS can be found in [12, 13, 14].

4 Parameter Synthesis

The system modeled in this work comprises two aircraft flying in opposite directions, as depicted in Figure 1. The models capture several kinds of maneuvers that the aircraft can perform in the same airspace. The automaton model for the aircraft that is flying from left to right is more complex than the model for the other aircraft, which is flying in the opposite direction. This is because, at some instants, the first aircraft may decide to engage in some kind of specific maneuver, while the second aircraft is assumed to remain cruising en route. The full system model comprises three individual hybrid automata: one for each aircraft and another one for the system controller.

All the case studies reported here focus on the overall system safety. For each case, an analysis of the model is conducted and some parametric synthesis of critical values, such as the aircraft relative height and horizontal separation, is also performed. By a parametric analysis, using symbolic constants, it is possible to obtain safe operational intervals for certain critical system parameters.

4.1 The Hybrid Automata Models

The model for the leftmost aircraft is depicted in Figure 1, and the model for the rightmost aircraft is shown in Figure 1. Variable x_i indicates the horizontal distance and variable y_i indicates the vertical position of the aircraft. Variable k is used to extract information about the direction of flight, as will be explained later. The $x_1 = x_2$ horizontal mark is the position where both aircraft cross, and it is called the *critical point*.

Note that the absolute position of the critical point varies, since it depends on the relative horizontal speed of both aircraft. Hence, the critical point is not always at the

³ 32000 feet \simeq 9754 meters; 30000 feet \simeq 9144 meters; and 2000 feet \simeq 610 meters.

⁴ 550 miles per hour \simeq 280 meters per second; and 490 miles per hour \simeq 250 meters per second.

⁵ <http://www.air-transport.org/public/handbook/>

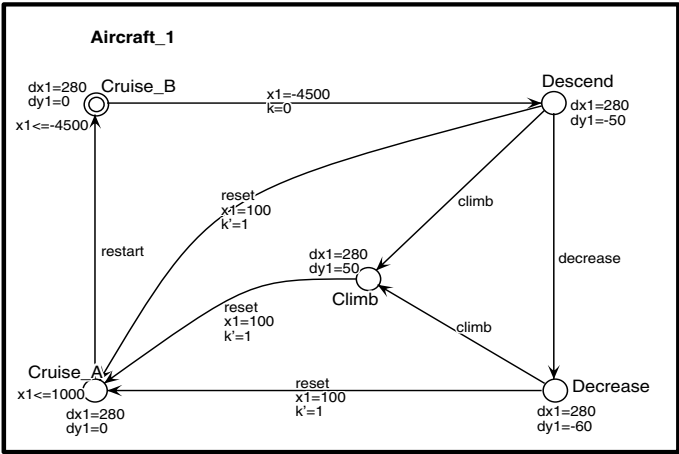


Figure 3. A model for the leftmost aircraft

zero horizontal mark. A negative value for the horizontal position is measured to the left of the zero horizontal mark, while positive values indicate distances to the right of this point.

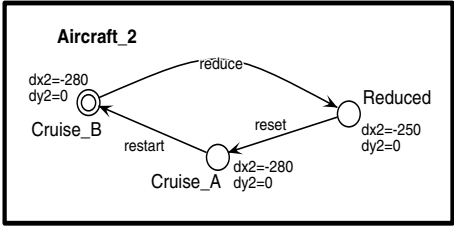


Figure 4. Aircraft_2 automaton

At the beginning, both aircraft are symmetrically positioned at 6000 meters from the zero mark. The leftmost aircraft is cruising at 9750 meters and the rightmost one is cruising at 9140 meters. When the horizontal separation between the aircraft is 9000 meters, the leftmost aircraft shifts to a descending trajectory. When the horizontal separation reaches 7000 meters, the TCAS on both aircraft start to interact.

The automaton for the leftmost aircraft starts off in the “Cruise_B” mode, as shown in Figure 1. From this mode, at the mark of 4500 meters, it enters in the “Descend” mode and starts to lose altitude at a rate of 50 meters per second. From this point, a first nondeterministic alternative calls for a scenario of no interaction between both aircraft. This is captured by the transition that goes directly from the “Descend” mode

⁶ About 10000 feet per minute.

to the “Cruise_A” mode. This transition happens at 100 meters to the right of the zero mark and it uses the “reset” event to synchronize with the models for the other aircraft and the controller. While in the “Descend” mode, the leftmost aircraft faces two other nondeterministic alternatives. It might receive a command to climb, moving into the “Climb” mode, synchronizing on the “climb” event, or it might receive a command to increase its descending rate, entering the “Decrease” mode and synchronizing on the “decrease” event. At the “Decrease” mode, there are two nondeterministic choices. Either, as a result of the interaction of both TCAS, the aircraft receives a counter-order to start climbing up, moving to the “Climb” mode, or it might fly by the second aircraft while still descending. In the first case, the leftmost aircraft will fly by the second aircraft while still climbing up. In any case, from the “Climb” mode or from the “Decrease” mode, a transition to the “Cruise_A” mode is taken, at a horizontal position of 100 meters. In both cases, a “reset” event is issued, in order to synchronize this move with the model that describes the behavior of the rightmost aircraft. Finally, at the “Cruise_A” mode, the first aircraft assumes a leveled flight and returns to the start mode no later than when it passes by the 1000 meters mark, to the right of the zero mark. The horizontal speed for this aircraft is kept constant at 280 meters per second⁷.

The second aircraft is modeled by a simpler automaton, composed of three modes, as shown in the Figure 1. The automaton starts off in the “Cruise_B” mode, positioned at 6000 meters to the right of the zero horizontal mark and cruising at 9140 meters. Initially, the aircraft horizontal speed is 280 meters per second, traveling from right to left. From this mode the aircraft will, upon detecting the “reduce” event, move to the “Reduced” mode, where its horizontal speed decreases to 250 meters per second. The aircraft travels at this speed until it detects the “reset” event and synchronizes with the automaton that controls the first aircraft, moving to the “Cruise_A” mode. Next, the automaton returns to the start mode, upon detecting the “restart” event.

The model of the controller implements (a simplified version of) the TCAS protocol, and is illustrated in Figure 1.

The automaton starts off in the “Normal” mode, and it may stay there until the horizontal separation between the two aircraft drops to 6000 meters. But earlier, at 7000 meters of horizontal separation, a nondeterministic choice occurs. Either the controller remains in the “Normal” mode, or it switches to the “Descend” mode. In the latter case, a “decrease” event occurs if the vertical separation between the aircraft is at least 400 meters⁸, forcing first aircraft to increase its rate of descent.

If the nondeterministic choice is not exercised at 7000 meters, the controller continues in the “Normal” mode, monitoring the horizontal and vertical separation between both aircraft. When the horizontal distance drops to 6000 meters, and if the vertical separation is at least 300 meters⁹, a “climb” event forces the first aircraft into a 50 meters per second¹⁰ climbing trajectory, since it is not safe to continue the descent and cross the second aircraft route. As a result of this, the controller reaches the “Climb” mode.

If the nondeterministic choice is taken at the mark of 7000 meters, then the controller may stay in the “Descend” mode until the leftmost one has moved up to 1000 meters to the right of the zero point. In the “Descend” mode, one of three other nondeterministic choices will prevail. First, the controller may issue a “reset” event and move

⁷ About 550 miles per hour.

⁸ About 1315 feet.

⁹ About 985 feet.

¹⁰ About 10000 feet per minute.

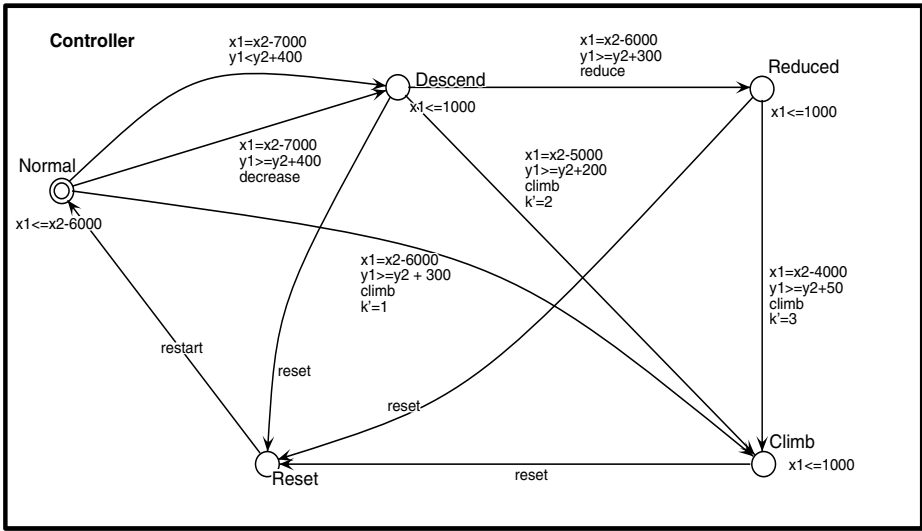


Figure 5. Controller automaton

into the “Reset” mode, where it stays until the first aircraft has passed the zero mark by 100 meters. Or the controller might decide to act when the horizontal separation between both aircraft has reached the mark of 6000 meters, their vertical separation being at least 300 meters, or the controller might act when the horizontal separation between the aircraft has reached the mark of 5000 meters, with the vertical separation between them being at least 200 meters. In the latter case, a “climb” event is issued, forcing the first aircraft in an ascending trajectory, and moving the controller to the “Climb” mode. In the first case, a “reduce” event slows down the rightmost aircraft and puts the controller into the “Reduced” mode.

After the controller reaches the “Reduced” mode a similar scenario evolves, with the controller monitoring the vertical and horizontal separation between both aircraft. The controller may stay in this mode until the aircraft have crossed and the first one has passed the zero point by 100 meters. Or, at a horizontal separation of 4000 meters, if the vertical separation between both aircraft is at least 50 meters, a “climb” event is issued and the controller passes to the “Climb” mode.

From the moment that the “Climb” mode is entered, the controller stays there until a “reset” event happens, when the first aircraft has passed the zero point by 100 meters. See also Figure 4. At this moment, the controller jumps to the “Reset” mode. In this mode, the controller waits until the leftmost one has traveled a distance not exceeding 1000 meters from the zero point. At the “Reset” mode, the automaton waits for the “restart” event and moves on to the “Cruise_B” mode, restarting the cycle.

The automaton that governs the overall system behavior is obtained by calculating the product of the three individual automata.

4.2 Case Studies

The system behavior was captured and analyzed by running a number of different experiments. All case studies discussed in the sequel focus on establishing the system safety. To this end, several parametric synthesis were run. Experiments of this kind can reveal the limits of a safe system operation and can also be instrumental in obtaining tighter values for a number of system parameters.

All experiments were run using the HyTech computational support tool¹¹. Figure 6 illustrates a typical input to the HyTech tool for a parametric analysis.

```

var final_reg, init_reg, reached: region;
init_reg:=   loc[Aircraft_1] = Cruise_B &
             loc[Aircraft_2] = Cruise_B &
             loc[Controller] = Normal &
             x1=-6000 & x2=6000 & y1=9750 & y2=9140 & k=0;
final_reg := loc[Aircraft_1] = Decrease &
             loc[Aircraft_2] = Reduced &
             loc[Controller] = Reduced &
             x1=x2 & y1<=height;
reached:=   reach forward from init_reg endreach;
print omit all locations
           hide non_parameters in reached & final_reg endhide;

```

Figure 6. Height parametric analysis

The region of initial configurations, *init_reg*, indicates that:

1. *Aircraft_1* and *Aircraft_2* are in their respective start modes, “Cruise_B”;
2. The *Controller* is in its start mode, “Normal”;
3. The initial position for both aircraft are specified next, in meters¹²;
4. The auxiliary variable *k* is zeroed. This variable indicates the mode of origin when the “Climb” mode is entered.

The parameter values used in this exercise are typical. They could be changed in order to reflect different values. An example of a final region, *final_reg*, capturing an unsafe region according to the TCAS protocol, is described next in Figure 7. It has five lines:

1. *Aircraft_1* is in the “Decrease” mode, indicating a steep descent;
2. *Aircraft_2* is in the “Reduced” mode, indicating a reduced horizontal speed;
3. The *Controller* is in the “Reduced” mode, reflecting the two conditions above;
4. Both aircraft are at the same position, that is, they are passing by each other;
5. The last condition parameterizes the vertical distance of the leftmost aircraft.

The next line, in Figure 7, asks the HyTech tool to perform a forward analysis in order to compute the *reached* region. The last four lines specify a parametric print out of the region of points belonging to *reached* and *final_reg* regions. This intersection will

¹¹ <http://www-cad.eecs.berkeley.edu/~tan/HyTech/>

¹² 6000 meters \simeq 3.3 miles; 9750 meters \simeq 32000 feet; and 9140 meters \simeq 30000 feet.

contain all unsafe points for the system, since *final_reg* describes the region of unsafety. By negating the region calculated by the tool, it is possible to obtain the region of safe values for the *height* parameter.

An unsafe condition of the TCAS operation means that for the values and restrictions imposed, obeying a certain flight plan, exists a violation in the performed maneuver between aircraft, reporting a safety limit for the maneuver.

All computational results were obtained by running the HyTech tool on a typical 350MHz Pentium II PC, with 320MB of main memory. Memory usage was never a problem when running the experiments. Also, for each experiment, the time consumed to complete the analysis was always quite reasonable, in the range of a few seconds, with the *Post* operator converging after a few iterations. In contrast, any attempt to increase the number of aircraft caused both the forward and backward analysis to fail. These observations indicate that the HyTech tool was operating close to its limit, and a careful construction of the models had to be undertaken, always working at these limits.

In all experiments, command line options were used in order to avoid library overflow errors, caused by multiplication of large integers. The output message “*Will try hard to avoid library arithmetic overflow errors*”, produced when running the HyTech tool, indicates that such options are active. Note that this message, by itself, does not say that the tool encountered overflow errors. When it did, the run were aborted and the results were discarded. Another point to be mentioned is that all backward analyses failed due, again, to multiplication overflow.

In the rest of this section, five case studies report on the synthesis of values for several parameters. These case studies were selected as representative of what could be achieved by a formal analysis of the TCAS protocol, using the HyTech tool. More details about the experiments can be found in [5].

The Minimum Height Before the Controller Acts. This study focuses on the minimum height reached by the first aircraft, in the worst scenario and before the controller starts to enforce the TCAS protocol. To capture this situation, the final region is specified by positioning the automaton for the first aircraft in the “Descend” mode, the automaton for the second aircraft in the “Cruise.B” mode, and the automaton for the controller in the “Normal” mode. Note that, because of the synchronizing “reset” event, it is not possible for the controller model to complete a full cycle before the aircraft models also restart at the initial mode. The output of the HyTech tool for this experiment is shown in Figure 7. The parametric analysis shows that the first aircraft can reach a minimum height of 9482 meters before the controller starts to act. Observe that the vertical separation is now $9482 - 9140 = 342$ meters.

```
Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 7
  7height >= 66375
Time spent = 0.63 sec total
```

Figure 7. Parametric analysis before the controller action

¹³ 9482 meters \simeq 31110 feet; 342 meters \simeq 1033 feet.

At the Critical Point with Increased Descend. This case study investigates the safety condition at the critical point when the first aircraft has taken the decision to increase its descend, while the other aircraft is still unaffected. The situation is specified by letting the automaton for the first aircraft enter the “Decrease” mode, while keeping the second automaton in the initial “Cruise_B” mode. The automaton for the controller moves to the “Descend” mode, forced by the “decrease” event, and stays there. See Figures 7 and 8. Introducing the extra condition $x_1 = x_2$ in the input file guarantees that the final region is taken at the critical point.

A synthesis for the minimum height value reached by the first aircraft yielded 8821 meters, or 28940 feet. From the point of view of the second aircraft, which is cruising at an altitude of 9140 meters, or 30000 feet, this value is inferior to the minimum acceptable distance of 2000 feet, as required by the protocol. However, from the point of view of the first aircraft, which is now below 29000 feet, the synthesized minimum value of 28940 feet is still within the acceptable interval of up 1000 feet of vertical separation.

In the same situation, that is, with the three automata reaching the same final modes, another study was conducted. Note that the controller automaton passes from the “Normal” to the “Descend” mode when the horizontal distance between both aircraft is 7000 meters, and the vertical distance between them is at least 400 meters¹⁴. In this second exercise, the 400 meters separation was parameterized by the *diff_height* variable. This was accomplished by changing the condition $y_1 \geq y_2 + 400$ into $y_1 \geq y_2 + \text{diff_height}$, in the controller automaton, shown in Figure 8. The HyTech tool, then, synthesized the maximum vertical separation taken at the moment when the first aircraft starts its increased descent, given that this change of behavior takes place exactly when the aircraft are 7000 meters apart. The result produced by the tool appears in Figure 8. It indicates

```
Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 4
7diff_height = 3020 & 7height >= 61750
Time spent = 0.15 sec total
```

Figure 8. Parameterizing vertical distances

that the maximum vertical distance between the aircraft can be 431 meters¹⁵. The minimum height reached by the first aircraft, at the crossing point, is still 8821 meters. This value shows that the vertical separation between the aircraft could be relaxed to 431 meters, in this situation.

It is also as easy to parameterize the horizontal separation between both aircraft, while maintaining the minimum vertical separation at 400 meters between them, both measured at the point when the first aircraft starts to increase its descent. Figure 9 presents the output of the HyTech tool for this exercise, where the parameter *diff_horiz* indicates the desired minimum horizontal distance. The synthesis revealed that the command to increase the descent should be issued no later than the point where the horizontal separation is 6648 meters. In this case, observe that the height reached by the first

¹⁴ About 1310 feet.

¹⁵ About 1414 feet.

```

Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 4
    diff_horiz = 6648    & 7height >= 61794
Time spent = 0.15 sec total

```

Figure 9. Parameterizing the horizontal and vertical distances

aircraft is now slightly higher, reaching 8827 meters¹⁶, a dangerous situation, but still an acceptable interval of vertical separation.

At the Critical Point and Climbing. Next, the situation where the first aircraft is ordered to climb is analyzed. In this case study, the first aircraft does not increase its descent, nor does the second aircraft reduces its cruising speed. The final region for the models is given by requiring the automaton for the first aircraft to reach the “Climb” mode directly from the “Descend” mode. The automaton for the second aircraft remains in the initial “Cruise_B” mode, and the controller automaton moves directly from the “Normal” mode to the “Climb” mode. All the evaluations are still being taken at the critical point. Note that, here, the value $k = 1$ is also part of the critical region. This guarantees that the automata are following the transitions as specified. See also Figure 1 and 2.

The result shows that the maximum height reached at the critical point, while the first aircraft is climbing, measures 10017 meters¹⁷. Note that this height is greater than the initial cruising height of 32000 feet. This shows that aborting the descent, here, could be premature, but is safe. The next experiments provide tighter values for these parameters.

First, the vertical separation of 300 meters is parameterized, on the transition from the “Normal” mode to the “Climb” mode, in Figure 3. The parameter *diff_height* replaces the value 300 meters in the condition $y_1 \geq y_2 + 300$. The value of 342 meters is returned by the HyTech tool for the vertical separation parameter. Since the horizontal distance between both aircraft was maintained at 6000 meters, the maximum vertical distance reached by the first aircraft while climbing was still the same 10017 meters.

When parameterizing the horizontal separation between both aircraft, at the point where the climb command is issued, the results returned by the tool are as shown in Figure 4. The vertical separation is maintained at 300 meters. As can be seen, now a linear relationship holds between the horizontal separation, when the maneuver is aborted by the climb command, and the vertical distance reached by the first aircraft, at the crossing point. The condition $diff_horiz \leq 9000$ is trivially observed, given that the first aircraft starts to descend when the horizontal separation between them is exactly 9000 meters. The linear relationship is, then, reduced to $(28height \geq 5diff_horiz + 250500) \wedge (diff_horiz \geq 5528)$. The last clause, $diff_horiz \geq 5528$, indicates that the minimum horizontal separation between the aircraft can be reduced to 5528 meters. In that case, the other clause says that the maximum climbing point, for the first aircraft, can reach 9933 meters¹⁸. This shows that, even if the first aircraft goes into a descent and

¹⁶ About 28960 feet.

¹⁷ About 32865 feet.

¹⁸ About 32588 feet.

```

Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 8
  28height >= 5diff_horiz + 250500    & diff_horiz <= 9000
    & diff_horiz >= 5528
Time spent = 0.21 sec total

```

Figure 10. Horizontal separation and climbing

then the maneuver is aborted at its minimum possible horizontal separation distance, the first aircraft can still reach a point higher than its original cruising altitude. In the worst case, the vertical separation between both aircraft, at the crossing point, is greater than 32000 feet and, as a consequence, the maneuver is deemed a safe one.

Increasing the Descent, then Climbing. This is the same situation as in the previous experiment, except that the first aircraft has already engaged in a steeper descent route. Here, the final region is modified only by changing the condition on the k variable. Note that the vertical separation at the point where the normal maneuver is interrupted by the climbing command is now at 200 meters, while the horizontal separation between both aircraft is set at 5000 meters, as dictated by the TCAS protocol.

The maximum height reached by the first aircraft, while climbing in this situation, is 9803 meters¹⁹. Observe, that this height is still greater than the initial cruising height of 32000 feet, but the difference is now smaller, when compared to the height reached in the maneuver studied in the previous experiment.


Next, the vertical separation of 200 meters was parameterized. The horizontal separation, in this case, remained at 5000 meters mark. The relaxed maximum value obtained was 217 meters, at the point where the climb command is issued. As in the

```

Will try hard to avoid library arithmetic overflow errors
Number of iterations required for reachability: 7
  56height >= 11diff_horiz + 494000    & diff_horiz >= 4840
    & diff_horiz <= 7000
Time spent = 0.18 sec total

```

Figure 11. Horizontal separation and vertical distance with increased descent, followed by a climb

previous test cases, the horizontal separation between the aircraft, when the climb command is issued, was also synthesized. The vertical separation was maintained at 200 meters. Figure  illustrates the output of the HyTech tool for this case. The relation $diff_horiz \leq 7000$ is trivially observed, since the first aircraft starts its descent already at the 7000 meters mark. Ignoring this clause, the conjunction computed by the tool reduces to the linear relationship $(56height \geq 11diff_horiz + 494000) \wedge (diff_horiz \geq$

¹⁹ About 32163 feet.

4840). Hence, the maneuver can be aborted as late as when the horizontal separation between the aircraft reaches 4820 meters. Even in this extreme case, the height reached by the first aircraft, at the crossing point, is 9772 meters²⁰, still allowing for the minimum of 2000 feet required for a safe operation.

Increasing the Descent, Reducing the Horizontal Rate and Aborting. In the last experiment, the first aircraft starts its descent, at a vertical rate of 50 meters per second, then it maneuvers to increase its downward vertical rate to 60 meters per second. The second aircraft, sensing the presence of the other aircraft, reduces its horizontal rate from 280 meters per second to 250 meters per second. When both aircraft are 4000 meters apart, the maneuver is aborted by the controller and the first aircraft receives a climb command.

The HyTech tool computed that the first aircraft reaches a minimum height of 9615 meters²¹ at the crossing point. Note that this value is approximately 1500 feet above the cruising altitude of the second aircraft. By the TCAS protocol, this would configure an unsafe operation scenario. This is because a minimum vertical separation of 2000 feet is specified when cruising at altitudes above 29000 feet, and this minimum is reduced to 1000 feet when cruising below 29000 feet. One alternative would be to use the HyTech tool and specify a final region where the vertical distance reached of the first aircraft, at the crossing point, was exactly 32000 feet, the minimum required for safety, and parameterize the horizontal separation required to reach that final region. This would yield the minimum safe value for that parameter.

In a final case study, the vertical separation between the two aircraft was also parameterized, at the moment when the “climb” event is issued, while the horizontal separation was kept at 4000 meters. The tool showed that the vertical separation can be at most 36310/37, or about 97 meters, for the “climb” event to be issued. The vertical separation at the crossing point was still an unsafe 9615 meters, or about 31546 feet.

When the HyTech tool was used to compute the relationship between the horizontal separation, at the point where the “climb” event occurs, and the vertical distance reached by the first aircraft at the crossing point, it was unable to complete the computation, due to the presence of multiplication library overflow errors.

5 Conclusions

An air traffic control protocol is a critical, real-time and reactive algorithm. It is clear that a possible system malfunction may cause enormous damages. The development of such systems, therefore, demands the application of a rigorous validation and verification procedure. This work describes a step in this direction.

Hybrid automata is the mathematical approach used to construct the models for the various real system agents studied in this work. The hybrid automata formalism allows for such agents to manifest a continuous, dynamic behavior, regulated by the asynchronous occurrence of discrete events. An Air Traffic Management (ATM) system, when using the Traffic alert and Collision Avoidance System (TCAS) protocol, exhibit all these characteristics. The cruising aircraft comprise the cooperating continuous dynamic agents of the system. The discrete asynchronous events arise from the exchange

²⁰ About 32060 feet.

²¹ About 31546 feet.

of commands, issued by the TCAS protocol, when some participating aircraft engage in a collision avoiding maneuver. The formal models developed here were based on the TCAS protocol and they describe the operational behavior of two aircraft, cruising in opposite directions. The aircraft were allowed to perform different maneuvers, under the guidance of the TCAS protocol.

The hybrid automata models, built from the TCAS protocol description, were used to verify the overall system safety, when operating in a number of different scenarios. The HyTech (semi) automatic computational support tool was used to validate all case studies. The HyTech tool was also used to synthesize some critical values for the vertical distance, and the vertical and horizontal separation between the cruising aircraft. The values obtained indicated which distances offer a safe operational scenario and, in some cases, indicated tighter values for some of these parameters. Although hindered, at times, when the complexity of the models reached the operational limits of the tool, most of the case studies were successfully planned and run on a typical desktop PC. More realistic cases studies, involving more aircraft and contemplating more complex maneuvers, could be attempted by adapting and extending the source code of the HyTech tool in order to make it run in larger and faster machines.

The HyTech tool proved easy to use, once the hybrid automata models were in place. Changing or adding new parametric variables was easily achieved from one case study to the next one. Describing new final regions, in order to capture different aspects of the system behavior, presented no difficulties either.

Related work, studying the safety of a subway system, can be found in [11].

References

1. Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual IEEE Real-Time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.
2. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a tool suite for automatic verification of real-time systems. Technical Report RS-96-58, BRICS, Aalborg University, DENMARK and Department of Computer Systems, Uppsala University, Sweden, December 1996.
3. Adilson Luiz Bonifácio, Arnaldo Vieira Moura, João Batista Camargo Jr., and Jorge Rady Almeida Junior. Análise e Verificação de Segmentos de Via de uma Malha Metroviária. In *Proceedings of the II Workshop on Formal Methods*, pages 13–22, Florianópolis, Brazil, October 1999. (In Portuguese).
4. Adilson Luiz Bonifácio, Arnaldo Vieira Moura, João Batista Camargo Jr., and Jorge Rady Almeida Junior. Formal Parameters Synthesis for Track Segments of the Subway Mesh. In *7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 263–272, Edinburgh, Scotland, 3-7, April 2000. IEEE Computer Society Press.
5. Adilson Luiz Bonifácio, Arnaldo Vieira Moura, João Batista Camargo Jr., and Jorge Rady Almeida Junior. Formal Verification and Synthesis for an Air Traffic Management System. Technical Report 05, Computing Institute, University of Campinas, Campinas, Brazil, February 2000.
6. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.
7. Kathryn T. Heimerman. Air traffic control modeling. As appears in the book entitled *Frontiers of Engineering 1997*, National Academy Press, 1998.
8. Thomas A. Henzinger and Pei-Hsin Ho. HyTech: The cornell hybrid technology tool. *Workshop on Hybrid Systems and Autonomous Control*, October 1994.

9. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HyTech. In E. Brinksma, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.
10. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. In O. Grumberg, editor, *CAV'97: Proceedings of the Ninth International Conference on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer-Verlag, 1997.
11. Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HyTech: Hybrid systems analysis using interval numerical methods. In Nancy Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144, Pittsburgh, April 2000. Springer-Verlag.
12. Pei-Hsin Ho. *Automatic Analysis of Hybrid Systems*. PhD thesis, Cornell University, August 1995.
13. John Law. Acas ii programme. ACAS Programme Manager, January 1999.
14. John Lygeros and Nancy Lynch. On the formal verification of the tcas conflict resolution algorithms. Technical report, Laboratory of Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, 1995.
15. Phil Scott. Technology and business: Self-control in the skies. *Scientific American*, pages 24–55, January 2000.
16. Claire Tomlin, George J. Pappas, and Shankar Satry. Conflict resolution for air traffic management: a study in multi-agent hybrid systems. In *IEEE Conference on Decision and Control*, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, 1997.
17. Lee F. Winder and James K. Kuchar. Evaluation of vertical collision avoidance maneuvers for parallel approach. In *AIAA Guidance, Navigation, and Control Conference*, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Crambridge, MA 02139, August 1998. Boston, MA.

Monitor-Based Formal Specification of PCI

Kanna Shimizu¹, David L. Dill¹, and Alan J. Hu²

¹ Computer Systems Laboratory, Stanford University, Stanford, CA 94305
`{kanna.shimizu,dill}@cs.stanford.edu`

`http://radish.stanford.edu/pci`

² Dept. of Computer Science, Univ. of British Columbia, Canada
`ajh@cs.ubc.ca`

Abstract. Bus protocols are hard to specify correctly, and yet it is often critical and highly beneficial that their specifications are correct, complete, and unambiguous. The informal specifications currently in use are not adequate because they are difficult to read and write, and cannot be functionally verified by automated tools. Formal specifications, promise to eliminate these problems, but in practice, the difficulty of writing them limits their widespread acceptance. This paper presents a new *style* of specification based on writing the interface specification as a formal monitor, which enables the formal specification to be simple to write, and even allows the description to be written in existing HDLs. Despite the simplicity, monitor specifications can be used to specify industry-grade protocols. Furthermore, they can be checked automatically for internal consistency using standard model checker tools, without any protocol implementations. They can be used without modification for several other purposes, such as formal verification and system simulation of implementations. Additionally, it is proved that specifications written in this style are *receptive*, guaranteeing that implementations are possible. The effectiveness of the monitor specification is demonstrated by formally specifying a large subset of the PCI 2.2 standard and finding several bugs in the standard.

1 Introduction

Given the importance of conforming to bus protocols, it is surprising that the current state of specification, even for widely used standards, is a long document written in English. Natural languages are ill-suited for precise specification because they tolerate vagueness, and are difficult for computers to analyze. The situation would be better if official standards, or even module interface specifications internal to companies, were written in a precisely-defined notation. However, until now, due perhaps to language and tool barriers, industry has largely ignored formally specifying interfaces. And so we present a specification *style* where the user need not learn any language beyond Verilog or VHDL and still be able to write formal specifications.

The style is based on writing the specification as a formal *monitor* (Figure 1). A monitor is an observer in a group of interacting modules, or *agents* which

communicate via a set of protocol rules. It's main task is to flag agents when they fail to uphold the protocol. Writing the specification as a monitor enables the specification to be written as a list of simple rules, thus, allowing formal specification to be a relatively easy process. It also allows the specification to be checked "stand-alone" where no implementation needs to be written to verify the protocol. Furthermore, it results in a synthesizable specification which can be directly used in testing environments where cycle-based models are needed. Another direct use is for modeling environments when model checking an implementation. And despite its simplicity, a monitor specification can be written for "real" protocols such as the widely-used PCI local bus protocol.

We also describe several highly effective debugging methods for monitor-style specifications. It is explained how certain requirements on the specification style discourages errors and how the debugging methods further ensure correctness and absence of contradictions. One highlight with a monitor specification is that debugging can be done on the protocol based on its internal consistency, before any implementations are designed. Furthermore, if two easy-to-check properties holds for the specification, it is guaranteed that the specification is receptive. Receptiveness guarantees that an implementation exists for the specification, and that there is no illusory freedom in the specification. On a practical level, these debugging methods found several problems in the official PCI protocol when they were applied to a specification of PCI.

The primary contributions of this paper are:

- *the definition of a simple yet powerful specification style that is resistant to specification errors;*
- *presentation of general specification debugging methodology, which does not require any implementations;*
- *a report on the successful application of the specification style and debugging methodology to PCI, and the resulting discovery of bugs in the protocol*
- *a theorem stating that the specification style together with a simple-to-check property, guarantees the receptiveness of a specification.*

Previous Work. Some of the same ideas were explored in a 1998 paper by Kaufmann, Martin, and Pixley [10], which proposed using logical constraints for environment modeling. All of the contributions listed above go beyond the Kaufman *et al.* paper. In 1999, Chauhan, Clarke, Lu and Wang [11] specified PCI using CTL and then model-checked the state machines that appeared in the appendix of the official PCI specification document [12] against their CTL specification. With our specification style, CTL is not used to specify the protocol; consequently, along with simplicity, our specification is executable and it can be used directly for a variety of applications, such as simulation, that a CTL specification cannot. In 1998, Mokkedem, Hosabettu, and Gopalakrishnan formalized and proved some higher-level properties of PCI involving communication over bus bridges [13]. Their specification is almost unrelated to the one here, which focuses on the low-level behavior of individual signals and ignores the higher-level transaction ordering properties which Mokkedem *et al.* concentrate on; the

difference is in the levels of abstraction of the specifications. Many specification languages for reactive systems and protocols have been proposed (too many to cite). However, it is important to note that we are specifically *not* proposing a new specification language, but a specification style that is simple enough to be implemented in a number of existing hardware description languages.

Notation. In the examples of the specification appearing in the paper, a logical notation is used. Individual variables (e.g. *frame*) are true when asserted and false when deasserted. This is a warning to readers who are accustomed to control signals being low when asserted. Logical connectives “ \rightarrow ”, “ \leftrightarrow ”, and “ \neg ” represent “IMPLIES”, “IFF”, and “NOT,” respectively. $prev(x)$ means the value of x in the previous cycle, $prev(prev(x))$ is the value two cycles ago. In an HDL, $prev(x)$ would be a state variable which is assigned the current value of x on each clock.

Structure of the Paper. Section 2 describes the specification style, what such a specification can be used for, and the rules for the style that ensures some of the correctness. Section 3 outlines the debugging methods and the bugs founds when applied to the PCI protocol. In section 4, the proof of receptiveness for a monitor specification is given. Section 5 is the conclusion.

2 The Monitor Specification

2.1 Description - The Specification Written as a Monitor

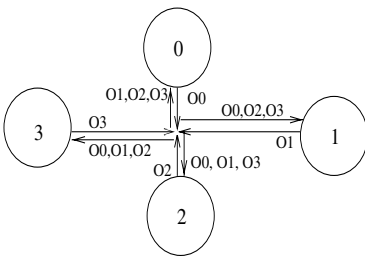


Fig. 1. The System View

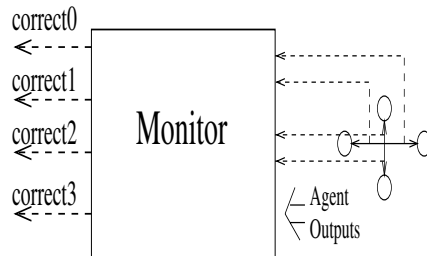


Fig. 2. The Monitor

A bus protocol specification can be viewed as a specification for a complete, closed system of agents using the bus. In Figure ■, agents 0,1,2,3 are using the bus and O0, O1, O2, O3 are the corresponding output sets. Because of the bus, the inputs for each agent are the outputs of other agents. (For agent 1, its inputs are O0, O2, and O3). A bus protocol specification dictates the

behavior of all the outputs relative to each other. A *monitor* that checks the agents' compliance to the protocol at each execution step, can be written. It is a machine with the agent output signals as its inputs, and boolean $correct_i$ signals as its outputs (figure 4). The monitor is such that as soon as an agent (or several agents) breaks the bus protocol, it singles out the erring agent(s) by making the corresponding $correct_i$ signal false. If $correct_i$ is true, agent i has upheld the specification so far and its current outputs also conform to the specification. If $correct_i$ is false, agent i has broken a specification requirement currently or sometime in the past. Thus, $correct_i$ is "sticky"; once a rule has been broken, the corresponding $correct_i$ stays false forever. *The specification style is based on writing the specification as such a monitor* 4. After all, the monitor must have exactly the protocol information to decide on agent compliance so it is natural for the protocol specification to be in the form of a monitor; it differs from the conventional view of a specification only because it is an active machine as opposed to a passive documentation. The immediate benefits of this are the direct applications of such a specification.

For Model Checking a Single Implementation. To verify a single agent implementation, one needs to create an environment for it, namely other agents on the bus. This is a non-trivial, tedious task. However with a monitor, an environment can be created without writing any implementation code. It does this by specifying which input sequences to the agent are correct according to the interface specification. Namely, one would model check the single agent by conditioning all the properties to be verified, with "if the interfacing agents have been correct so far according to the monitor". For example, if p is the property to be model checked, and agents i and j form the environment, the property to be model checked is " $correct_i \wedge correct_j \Rightarrow p$ " where $correct_i$ and $correct_j$ correspond to the output signals of the monitor for i and j . The monitor and the condition in the model checking properties correctly constrain the inputs to the agent. This is an example of assume guarantee reasoning where the specification for one (or more) agent(s) is used to verify the implementation of another agent. This use of the monitor is very similar to what is described in 5. As an execution example of this technique, Govindaraju used our PCI monitor to successfully verify a PCI controller implementation 6.

For Simulating Complete System Implementations. In a testing environment, an interface monitor, if written in the language of the implementation, can be directly connected to a design and flag errors and correctly assign blame to the erring module in a system-level simulation. Since monitors can be written in synthesizable RTL, they can be used for tools that need cycle-level models instead of event-based simulation models, such as formal verifiers or emulators.


¹ Only the monitor is written by the specification writer. The agents in the figure are to be later implemented by someone else.

2.2 Construction of a Monitor Specification

A further advantage of the monitor-style specification is that they are very easy to construct. First, it is noted that a specification is a list of rules. In particular, the official PCI 2.2 specification is written that way. Thus, it is natural for the monitor to check for each of these rules independently. For clarity, these rules will be called *constraints* here. Here are some examples of PCI constraints,

“TRDY# cannot be driven until DEVSEL# is asserted.” (section 3.3.1)
 “Only when IRDY# is asserted can FRAME# be deasserted” (section 3.3.1)

As logic formulas, these can be written as follows; $trdy \rightarrow devsel$ (if $trdy$ is true, then $devsel$ must be true) and $prev(frame) \rightarrow frame \vee irdy$ (if $frame$ is true in the last cycle, then it must either be true in this cycle or if it's not, $irdy$ must be true). The goal was to keep the constraints as simple as possible to prevent the overall specification from getting complicated. When specifying PCI, it was found that the following constraint characteristics can be kept true, and the specification can still fully describe the protocol.

1. No CTL or LTL. For the monitor specification, all of the PCI constraints can be written without using any CTL  constructs nor is knowledge of any linear time temporal logic (LTL) specifically needed. This is the basis for the claim that the specification style can be used with HDLs such as Verilog. In Verilog, the above example becomes, (where $correct_i$ is initialized to 1.)

```
if(trdy && !devsel) {correct = 0;}
```

2. No Complex State Machines. Only two types of very simple state machines were needed as auxiliary variables for the PCI constraints. One type is a event-recoding state machine which becomes true when a *set* event happens and remains true until a *reset* event occurs and is false otherwise. This is needed, for example, to “remember” whether the transaction is a read or a write. The second type is a counting state machine which starts to count after a *set* event, and stops counting either when a *reset* event happens or a *limit* is reached, whichever comes first.

3. Small Time Frames. With the help of the state machines described above, all of the constraints can be written with less than three time frames. Thus, the most complicated PCI constraint looks like this: $prev(devsel) \wedge prev(prev(stop)) \wedge prev(stop) \wedge prev(final_dphase_done) \rightarrow \neg req$. For most constraints, only two time frames are needed, and thus, most are as follows: $prev(stop) \wedge \neg prev(devsel) \wedge \neg prev(dphase_done) \rightarrow \neg devsel$. This property keeps the constraints compact.

From a preliminary inspection of a more complex protocol than PCI, such as Intel's Merced bus, properties 1 and 3 seem to hold for other protocols. Thus, a specification can be a list of compact constraints which are easy to maintain. And to construct the desired monitor, the constraints are directly used to determine

the $correct_i$'s. Assuming that each constraint constrains the behavior of only one agent, the constraints are grouped by the agent which they constrain. When the agent output signals make all the constraints of one agent true, the corresponding $correct_i$ is true; otherwise, the $correct_i$ is false. Thus, $correct_i$ is a conjunction of all the constraints specifying the behavior of agent i . The following is the assignment statement for $correct_i$, where $constraint_i^j$ pertains to agent i .

if $(constraint_i^0 \wedge constraint_i^1 \wedge \dots \wedge constraint_i^n)$
 then $correct_i = \text{true}$, else $correct_i = \text{false}$

Therefore, the monitor is a list of propositional formulas, auxiliary state variable assignments, and $correct_i$ assignments. There is no conversion of this to a state machine; this is precisely the code for the monitor.

(propositional formulas)
 $constraint_i^0 = trdy \rightarrow devsel$
 ...
 (state variable assignments and the two types of small state machines)
 $prev(trdy) = trdy$
 ...
 ($correct_i$ assignments)

2.3 Detailed Style Requirements for a Monitor-Style Specification

Some requirements on the constraints were discovered and developed. In this section, the motivation behind them will be discussed.

Separability of the Constraints Rule. *Each constraint can only constrain outputs of one agent.*

For each constraint, there should only be one agent to blame when the constraint is broken. Consequently, a constraint can only restrict the outputs of one agent; if it dictates the output behavior of two or more agents, multiple agents can be held responsible for a single broken constraint. Since it is exactly the current state variables that are constrained by the constraints, all current state variables of a constraint, must be outputs of the same agent. Equivalently, since for a particular agent, outputs of other agents are its inputs, the constrained variables must all be the agent's outputs and not its current inputs.

This rule is called the *separability* rule because it allows constraints for different agents to be separated and evaluated independently. This separability factor is important for a specification because with it, a specification can be guaranteed to have an implementation as proved in section 4. The main need for the separability of a constraint is to uphold an important principle of specifications: *it should be possible to implement an agent based on information solely from the specification, and know that the implementation can interact correctly with any other agent upholding the specification.*

Independent Implementability. If multiple agents are responsible for upholding an inseparable constraint, the implementation of a single agent must be able to do the impossible act of predicting the behavior of the other agents. Thus, the only way such a system can be designed is for the different agents to be designed *together* which runs counter to the above principle. For example, a bad, inseparable specification would be $a = b$, where a and b are outputs of two different agents. An implementer of one of the agents cannot safely set the value of a without knowing what the implementer of the other agent will set the value of b to. Such a functionality is not independently implementable. Equivalently, a monitor for such a specification cannot blame a single agent when the constraint is broken. If $a = b$ does not hold, it is impossible to decide whether a is wrong or b is wrong. Thus, it is apparent why “a broken constraint can only blame one agent” is a sufficient condition for the specification to uphold the above principle.

Removing Illusory Freedom. Consider the specification “ $\neg(a \wedge b)$,” where a and b are outputs of different agents. Such a formula might result from an attempt to specify mutual exclusion. In this case, an implementer can only set $a = 0$ safely, in case the implementer of the other agent may set $b = 1$. But the implementer of b can only safely set $b = 0$, too, so the specification could just as well be $a = b = 0$. In other words, this specification allows *illusory freedom*, which is also undesirable. The separability style rule disallows such situations.

Specifying Moore Machines. As the clock speeds of busses gets faster and faster, almost all bus interface protocols assume a Moore machine timing; namely, it is expected that the interfacing agent needs at least one cycle to respond to its inputs. The output separability rule can be thought of as a result of modeling the agents as Moore machines. For example, machine A has an input in_A and outputs o_A and r_A and is specified a constraint that breaks the separability style rule: $\neg in_A \vee o_A \wedge r_A$. This can be interpreted as whenever in_A is true, machine A must react immediately and assert its outputs o_A and r_A true. But since machine A is a Moore machine, this is an unreasonable specification.

The example of mutual exclusion is interesting because although it is frequently a desired property of a bus, “only one agent can be driving the bus at a time”, the style rule disallows it as a constraint as outlined in the “Removing Illusory Freedom” paragraph. Mutual exclusion is not a specification that can be implemented independently by several agents. Instead, it is an emergent property that is implied by other constraints that can be specified independently for the agents. For example, in PCI only the agent that is the current *master* can drive certain bus signals, which is a property that can be specified as a separable constraint. The arbiter in the system ensures that only one agent is the master at any time and this is also a separable constraint. Together, these separable constraints ensure the non-separable mutual exclusion property of the bus.

An Un-Implementable PCI 2.2 Requirement. Interestingly, there is an official PCI 2.2 specification  requirement which does not satisfy this separa-

bility rule, and is consequently un-implementable as stated. However, there is an equivalent, implementable requirement to replace it.

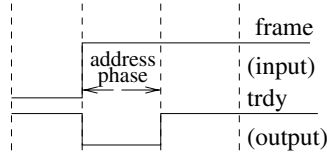


Fig. 3. *trdy* behavior during the address phase

The requirement, which is in section 3.2.4 of the official PCI documentation, states that the signal *trdy* must use the address phase as a turnaround cycle. The “address phase” is when the signal *frame* is asserted when it was de-asserted in the previous cycle (figure [i](#)), and a “turnaround cycle” for a bus signal is a cycle where no agent is allowed to drive that signal. Thus, the requirement translates to “if *frame* just became asserted in this state, do not drive *trdy*.” The problem is *frame* and *trdy* can be driven by different agents, and so both must decide simultaneously how to meet the requirement *together*. And a Moore machine agent cannot react (via the value of *trdy*) to its input (in this case, *frame*) in the same cycle. Thus, this requirement cannot be implemented in an agent as stated in the standard. However, this requirement can be stated in a different way with the same intended effect. The requirement should be “no agent may drive *trdy* if *frame* and *iridy* were both deasserted in the previous state.” This obeys the output separability rule and it will enforce the desired property that all agents not drive *trdy* in the address phase.

The Importance of Isolating Current Variables in a Constraint. *If the constraint spans more than one time frame (i.e. involves at least the previous state) the constraint must be written in the form “past_conditions → current_state”. Thus, all multi-state constraints must be written as implications, and all prior states variables must be in the antecedent and current state variables must be in the consequent.*

Example: ‘‘Only when IRDY# is asserted can FRAME# be deasserted.’’

Correct: $prev(frame) \rightarrow frame \vee irdy$

Incorrect: $prev(frame) \wedge \neg frame \rightarrow irdy$

Unlike the previous rule, this rule is not required for the implementability of the specification, but writing the constraints in this form makes the specification easier to understand and debug. This style rule separates the conditioning element, the *past* history, from the constraining element, the *current* constraint. Also, this form makes contradictions in the constraints easier to spot, as demonstrated in section [4.1](#).

² A note to PCI experts: for an address phase following a back-to-back transaction, which won't have *frame* and *iridy* deasserted in the previous state, other constraints ensure that there is a turnaround cycle for *trdy*.

3 Debugging the Monitor Specification

A very important practical question is how to debug a specification. The following debugging methods work with the monitor-style specification *without requiring any implementations to be written*. Once the specification is written, these methods can be immediately and directly applied to it. They check whether the monitor is overly restrictive where it flags correct actions as errors, or under-restrictive where it does not catch incorrect actions. Therefore, they check for contradiction and completeness, respectively.

This section also includes the bugs found by these methods in the monitor-style formal PCI specification. Some were translation errors which are significant because they support the claim that an informal specification is prone to misinterpretations. However more importantly, some inherent problems in the official PCI protocol were discovered. These discovered bugs further stress the importance of using a formal specification style to develop and review a protocol. For these debugging methods, a CTL model checker is needed. Good model checking tools exist (such as CMU's SMV [1] and Cadence's SMV [2] which were both used successfully with the PCI specification) that will take a monitor-style description and answer queries about the defined state graph. The queries are written in the branching-time temporal logic CTL [3].

3.1 Dead State Check

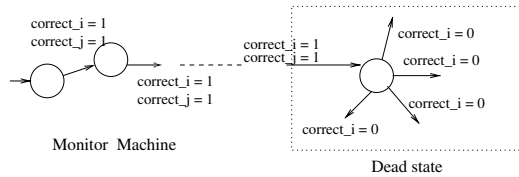


Fig. 4. A dead state for agent i

Dead states arise due to contradictions in the specification. For example, if one constraint for agent i requires p to be true in the current state and another requires $\neg p$ to be true in the same state, there is a dead state. Defining dead states precisely requires defining a few other concepts first. A transition in the monitor machine from a state is said to be *correct for agent i* if the monitor asserts $correct_i$ during the transition. A monitor state s is *correctly reachable*, if there exists a sequence of agent outputs that causes the monitor state to enter s from the initial state, while **all** $correct_i$ signals are continuously asserted. A dead state of the monitor for agent i is a correctly reachable state that has no correct exiting transitions for agent i ; for all outgoing transitions of the dead state, $correct_i$ is false. Intuitively, a good specification should have no correctly reachable dead

states because then, all possible agent outputs are incorrect according to the monitor. (Others have observed the importance of dead states [11].)

To ensure the absence of a dead state in a monitor specification, a certain characteristic needs to hold for all the agents in the specification: “for every state in the monitor where no constraints for any agent have been broken so far, there must exist at least one next state where all of the constraints for the particular agent hold”. This characteristic can be checked easily using a CTL model checker with the formula, for a particular agent i , $AG(\bigwedge_{j \in Agents} correct_j \rightarrow EX correct_i)$.

If there are any contradictions in the specification, the model checker for this property returns a counterexample indicating a dead state.

The following is an incorrect logical translation of some PCI requirements, a mistake actually made by the first author. The dead state check found a dead state which resulted from the conflicting constraints,

$$\begin{aligned} prev(address_phase) &\rightarrow \neg trdy \\ prev(trdy) &\rightarrow trdy \vee (irdy \wedge (stop \vee trdy)) \vee prev(irdy \wedge (stop \vee trdy)) \\ prev(trdy) &\rightarrow (prev(stop) \leftrightarrow stop) \end{aligned}$$

The contradiction is not obvious from the expression above but if it is re-written to obey the *Isolate Current Variables* rule, and the values of the state variables in the dead state are known from the dead state check, it is possible to see that there is no legal next state when $prev(address_phase \wedge \neg irdy \wedge trdy \wedge \neg stop)$ holds because $\neg trdy \wedge (trdy \vee (stop \wedge irdy)) \wedge \neg stop$ is unsatisfiable.

$$\begin{aligned} prev(address_phase) &\rightarrow \neg trdy \\ prev(\neg irdy \wedge trdy) &\rightarrow trdy \vee (stop \wedge irdy) \\ prev(trdy \wedge \neg stop) &\rightarrow \neg stop \\ prev(trdy \wedge stop) &\rightarrow stop \end{aligned}$$

Since the dead state check returns the dead state as “..., $address_phase = true, irdy = false, trdy = true, stop = false$, ...”, these variable assignments can be used to see which constraints are in effect, by plugging these values into the left-hand-side of the implications (the antecedents). In this case, the first three constraints are in effect and their consequents form the contradiction. This process is effective if the constraints follow the *Isolate...* rule and the dead state check can return the dead state’s state variable values. One advantage this check has over the other checks is its simplicity. No creativity or expertise is required; only the CTL formula given above and a model checker are needed.

Another similar check tests for under-restriction in the specification. It is reasonable to assume that in all states, at least one constraint is in effect. The check searches for correctly reachable states where *all* possible outputs for agent i are correct according to the monitor. The monitor can be checked for this property with the CTL formula, $EF((\bigwedge_{i \in Agents} correct_i) \wedge AX correct_i)$ “There is a correctly reachable state where *all* the possible next states for an agent are correct.” Although this check turned up no bugs in the PCI monitor specification, it is still a worthwhile check. Like the dead state check, this check requires no creativity or extra work.

The bus agents are assumed to be Moore machines in this paper but Bryant, Chauhan, Clarke, and Goel define and describe inconsistencies for combinational Mealy machine circuits in [10].

Results From Applying the Dead State Check to PCI. This check proved to be more effective in catching errors introduced in the monitor writing stage rather than serious problems in the protocol itself. Specifically, the dead state check pinpointed typos by the monitor specification writer, misinterpretations by the monitor writer due to the ambiguous wording in the protocol text, and exceptions to general rules not mentioned by the official specification. Because this check proved to be effective in finding the exact intent of the requirement, it proved indispensable for making the constraints precise. (See example below.) Four bugs in the formal specification which were due to misinterpretation of (arguably) ambiguous wording in the official documentation were found by dead state checking. Furthermore, the test aids the specification writer in realizing the boundary cases for general rules by demonstrating how a contradiction can occur in special cases. Thus, the dead state check helps the specification writer identify exceptions to the too generally-stated rules, *which are not mentioned by the official document*. The dead state check discovered six such cases where the monitor writer needed to refine the constraints to account for the special cases.

As an example of an ambiguously worded requirement which was misinterpreted and thus caused a contradiction in the monitor, consider the following from section 3.3.3.1: “IRDY# must remain asserted for at least one clock after FRAME# is de-asserted” which seemingly translates to the constraint

$$1. \text{prev}(\text{prev}(\text{frame})) \wedge \text{prev}(\neg \text{frame}) \rightarrow \text{irdy}$$

However, in section 3.3.3.2.1, it is stated that “the master must de-assert IRDY# the clock after the completion of the last data phase.”

$$\begin{aligned} \text{last_data_phase} &= \neg \text{frame} \wedge \text{irdy} \wedge (\text{trdy} \vee \text{stop}) \\ \text{prev}(\text{last_data_phase}) &\rightarrow \neg \text{irdy} \\ \Rightarrow 2. \text{prev}(\neg \text{frame} \wedge \text{irdy} \wedge (\text{trdy} \vee \text{stop})) &\rightarrow \neg \text{irdy} \end{aligned}$$

The conjunction of these two constraints causes a conflicting requirement on *irdy* in the correctly reachable state where both antecedents are true: $\text{prev}(\neg \text{frame} \wedge \text{irdy} \wedge (\text{trdy} \vee \text{stop}) \wedge \text{prev}(\text{frame}))$. In the next state, the first constraint states that *irdy* must be true and the second, *irdy* must be false. It was concluded from guessing at the intention of the requirement that the first rule was misinterpreted and the correct interpretation of it is $\text{prev}(\text{frame}) \rightarrow \text{irdy} \vee \text{frame}$ which admittedly is puzzling because this constraint doesn’t require the “one clock after” part of the requirement.

3.2 Characteristic Check

Another more powerful debugging method is checking for specific properties, or *characteristics* in the specification. These characteristics are mainly logical

statements about agent events. If the monitor is too constricting, certain agent actions which should be possible will not be allowed by the monitor. This method also catches loopholes in the specification which allow behavior that should be illegal, and so completeness of the specification can be gauged. This debugging method is not new but it furthers the case for formally specifying protocols and more importantly, it found several bugs in the official PCI protocol standard. These characteristics are expressed as CTL formulas and are checked against the monitor using CTL model checking. It must be emphasized that the *specification* constraints are simple, bounded, linear time properties and the *checking* characteristics are more complex, unbounded, CTL formulas. *This allows the specification to be synthesizable and yet guarantee rich properties.* One drawback of this characteristic checking is that a user must come up with the characteristic statements. They cannot be automatically deduced from the specification. It is also subject to false error reports when the characteristics themselves are incorrect.

Results From Applying the Characteristic Check to PCI. 114 characteristics were written in CTL and checked against the monitor-style PCI specification. This checking method found sixteen bugs in the monitor which resulted from errors in the monitor writing process, but more importantly, seven bugs in the official standard were found by this method. For finding actual problems in the official specification, this test proved to be more effective than the simple dead state check. Here are some characteristics that exposed problems in the official specification. These or similar characteristics are probably applicable for protocols other than PCI and are considered general system properties that should be checked for.

1. System Must Always, Eventually Return to the Idle State. It is reasonable to assume that the system should always be *able* to reset into the *idle* state; if there are any deadlocks states which forbid this from happening, checking for this characteristic should find such a problem. However, it is the stronger property, “the system must always *inevitably* reset and go back to the idle state” which found problems in the PCI protocol. This can be expressed in CTL as $AG \left(\bigwedge_{i \in Agents} correct_i \rightarrow \neg EG \left(\bigwedge_{i \in Agents} correct_i \right) \wedge \neg idle \right)$ where *idle* is defined as $(\neg irdy \wedge \neg frame)$. “If in a state where all agents have been correct so far, then the system must eventually reach a state where all agents are still correct and the bus is in the idle state.” This characteristic must be true for the PCI protocol because only when the bus is idle, can a new agent start a transaction. If the bus is never idle because one agent is constantly driving it, this agent has effectively taken over the bus never allowing other agents to use it. To avoid such a situation, the specification must force an agent to always, eventually relinquish the use of the bus as a master and let the bus state be idle.

There are three legal ways the PCI protocol allows an agent to not relinquish the bus. Essentially in all three cases, *frame* is deasserted while *irdy* is asserted by the agent (Figure 1). An agent can keep the bus in this state because of the

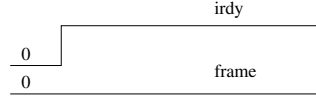


Fig. 5. A non-Idle State

following. There is a timer counter which counts the number of cycles *frame* has been asserted and when it exceeds a preset value, the specification requires the agent to deassert *frame*. Thus, the protocol intends to limit one agent's use of the bus by observing the assertion of *frame*. The protocol's shortcoming is in not recognizing that in an $irdy \wedge \neg frame$ state, $\neg frame$ keeps the timer counter deactivated but *irdy* keeps the bus non-idle.

1. From an idle state, the protocol allows an agent to assert *irdy* and remain in this non-idle-bus state ($irdy \wedge \neg frame$) forever. (Figure 5)
2. During the data phase of a single data phase transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a *trdy* or a *stop*, the agent can remain in this non-idle-bus state forever.
3. During the last data phase of a transaction, *frame* is deasserted and *irdy* is asserted. If a target doesn't respond with a final *trdy* or a *stop*, the agent can remain in this non-idle-bus state forever.

2. Definitions are Disjoint. Protocols allow an agent to communicate to other agents abnormal terminations when a transaction cannot be carried out. Usually, there are several different types of terminations and an agent asserts and de-asserts different bus signals to indicate which termination type it is executing. For example, in PCI, one termination type, *target_abort* is defined as $target_abort = \neg devsel \wedge stop$ and another type, a *retry* as $retry = stop \wedge \neg trdy \wedge initial_data_phase$ in section 3.3.3.2 of the documentation. The other agent involved in the transaction, namely the master agent, must react differently to each target termination type, so the ability to identify a termination type uniquely from the signals of the terminating agent is important. We can check whether these terminations are distinct by checking the CTL formula $AG \neg ((\bigwedge_{i \in Agents} correct_i) \wedge target_abort \wedge retry)$, ("There is never a state where the specification holds and *target_abort* and *retry* are signaled simultaneously") which reveals that there is a correctly reachable state where $\neg trdy \wedge stop \wedge \neg devsel \wedge initial_data_phase$ holds, which is consistent with either *retry* or *target_abort*. Therefore, the protocol allows an agent to signal both termination types simultaneously. If the PCI protocol had been originally written in a formal form and tested for this characteristic, this ambiguity in the protocol could easily have been found and resolved before the protocol became a public standard.

3. Termination Types Should not Change During a Single Transaction. The third characteristic checks whether termination types can change during one

transaction. For example, it checks whether an agent can signal a *target abort* in one clock cycle and then a *retry* in the next clock cycle before the transaction ends. Amazingly, checking the property $AG \neg ((\bigwedge_{i \in Agents} correct_i) \wedge target_abort \wedge EX((\bigwedge_{i \in Agents} correct_i) \wedge retry))$ (“This never happens: the specification holds so far when a target abort is signaled, and there is a possible next state where the specification still holds and retry is signaled”) reveals that PCI allows this. It is ambiguous, for example, how the master agent should be implemented to react to a target which signals a *target abort* which indicates that the target is not capable of handling the requested data but then, signals a *disconnect with data* which allows the target to transfer data.

4 The Receptiveness Proof

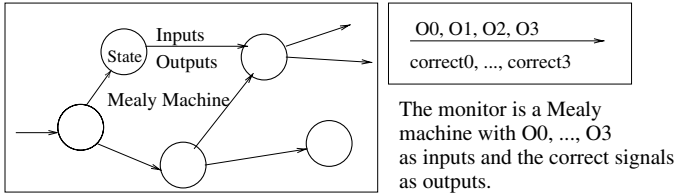


Fig. 6. A Mealy Machine

A Mealy machine has its inputs *and* its outputs on its edges; the output is not associated with a state as with Moore machines. A monitor is a Mealy machine because in order to determine the output $correct_i$ values, a combinational function on the input observed signals and internal state variables is sufficient (Figure 4, 4).

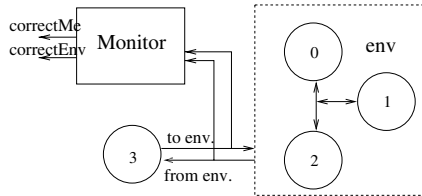


Fig. 7. One Agent and the Environment

One can view the system of agents such that one agent is an object of interest and the other agents form its environment. Using the same monitor, we now have outputs $correct_{me}$ for the former and a $correct_{env}$ for the latter. ($correct_{me}$ can

be a particular $correct_j$ and $correct_{env}$ would be a logical conjunction of $correct_i$ for $i \neq j$.) (Figure 4) This setup can be viewed as a game between the agent of interest, me , and the environment, env . Agent me and env output signals in a locked synchrony, and do not alternate driving an output. It is deemed that as soon as env breaks a specification rule ($correct_{env}$ becomes false), me has “won” and the monitor’s $correct_{me}$ value will remain true regardless of me ’s outputs. This is a reasonable restriction because an agent should not be required to uphold the specification if the environment has fed it illegal inputs. It will be proven that if two requirements hold for the specification of the system, it is guaranteed that a Moore machine K exists where no matter what the environment outputs to it, K will always output signals that will keep $correct_{me}$ true; K implements the specification. With such a K , the environment will never be able to force K to output an illegal sequence.

The first requirement on the monitor is the output separability rule (section 4.1) which is restated here as “the function which determines $correct_i$ must only be a function of the current state of the monitor machine and the current output of agent i , and not the current outputs of any other agent.” The information of the output values of the other agents is incorporated into the next state of the monitor machine. The second requirement is that there are no dead states (section 4.2) for agent me in the monitor. For every correctly reachable state of the monitor, there is at least one transition out of that state with the $correct_{me}$ on the edge as true.

Theorem 1. *If a Mealy machine monitor, M , which obeys the following requirements exists for some specification, then a Moore machine implementation for the single agent me is guaranteed to exist. The restrictions are,*

1. *The monitor must not have any dead states for agent me .*
2. *The monitor must observe the output separability rule.*

And it is assumed that once the environment breaches the specification, $correct_{me}$ is infinitely true.

Proof Sketch: Because of assumption 2, a correct output for the agent can be determined at every state independent of the current input. Assumption 1 guarantees the existence of a correct output for me for every correctly reachable state.

Proof: Please see Appendix A for the full proof.

If one views this system once again as a multiple agent model (Figure 4), an interesting corollary can be deduced from Theorem 1.

Corollary 1: A Set of Implementations Exist for a Specification *If a Mealy machine monitor, M , which obeys the following restriction exists for some specification, then a set of Moore machines which implement the specification is guaranteed to exist. The restrictions are,*

1. *The monitor does not have any dead states for all agents in the specification.*
2. *The monitor observes the output separability rule.*

Proof Sketch Apply theorem 1 to each agent in the specification and the theorem guarantees a correct implementation of all agents. Since the theorem guarantees specification compliance, independent of the inputs to the agent, the agents can be implemented independently and be guaranteed correctness when composed together.

An implementation can always choose the left branch in the specification to avoid the dead state. Receptiveness disallows such illusory freedom in the specification.

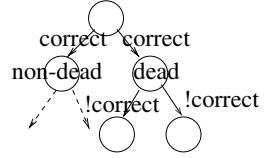


Fig. 8. How even with Dead States in the Specification, an implementation can exist

Corollary 2: Receptiveness A monitor specification is defined to be *receptive* ■ if for every correctly reachable state in the monitor, there exist agent implementations, when connected to each other and to the monitor, can cause the monitor to reach that state. ■ Receptiveness ensures that there is no illusory freedom in the specification. For example,

$$\begin{aligned} prev(a) &\rightarrow out_0 \vee out_1 \\ prev(out_1) &\rightarrow \neg c \\ prev(out_1) &\rightarrow c \end{aligned}$$

out_0 and out_1 are outputs of one agent and so the agent can always choose to assert out_0 instead of out_1 to avoid the inevitable error state caused by asserting out_1 (because of the last two constraints). Thus, even with a dead state in the specification, there exists an implementation; this example illustrates how the absence of dead states is not a necessary condition for Theorem 1. Receptiveness is a tougher condition to satisfy than implementability, and ensures that there is no illusory freedom in the specification such as “ $\vee out_1$ ” in the first constraint. Every correctly reachable state must have a correct next state in order for receptiveness to hold for a specification.

A specification is receptive if

1. The monitor does not have any dead states for all agents in the specification.
2. The monitor observes the output separability rule.

Proof Sketch: State s is a correctly reachable monitor state and the sequence of correct agent i outputs $\{O_0^i, O_1^i, \dots, O_n^i\}$ which lead to state s is known for $\forall i$ in *Agents*. These agents are individually constructed such that they output this sequence. Thus, the set of agents can take the monitor to state s ; it remains to be shown that the agents implement the specification, namely, that their outputs keep the $correct_i$ ’s true. Up till state s , it is obvious that the agents

³ Ed Clarke has also recognized the relevance of receptiveness to bus specifications, but proposes using model checking algorithms that can check the property directly.

are correct because the output sequences were chosen along the edges where all $correct_i$'s are true. Such a sequence exists because s is correctly reachable. As for after state s , there exists a next state s' such that the transition from s to s' is *correct*, because of assumption 1. The outputs along this transition can be used for the agent implementations. Inductively, it can be shown that at each step, a correct output can be independently chosen for all agents because of the assumptions. Thus, the agents implement the specification.

5 Conclusion

The monitor-style specification of PCI consists of 83 rules. The entire description file has 280 lines excluding comments. The specification covers almost all signal-level requirements from section 3.1 to section 3.6 of the official 2.2 PCI specification documentation [1]. Bus bridges and 64 bit extensions are not included. The PCI formal specification was fully debugged using the different checks introduced in section 4. There are no dead states in the current specification and all characteristics written, hold. Most of the model checking was done using Cadence SMV [2] on a Pentium Pro system with 128M of memory where the model checking runs took under 5 minutes and model checking the specification was not a problem. The monitor-style PCI specification, written in Verilog, is available at <http://radish.stanford.edu/pci>.

An obvious next step for this line of work would be to attempt specifications of other interfaces, especially those with characteristics different from PCI, such as pipelined busses. It is likely that different stylistic issues will arise, as well as complexity issues. A second direction is to find additional uses for monitor specifications, to maximize their value. One possibility would be interface synthesis for which work has been started by Clarke, Lu, Veith, Wang, and German [3]. A more modest goal would be to extract *don't cares* from the specification to aid in optimizing a synthesizable description of an interface implementation. Another possibility is to use the monitor as an activator which generates test vectors for an implementation automatically from the monitor description, or as a checker to measure testing coverage. This is to quantify the amount of testing that needs to be done in order for the implementation to be considered thoroughly tested. A third direction of interest is to develop better tools for debugging the specification. For example, the dead state check is limited in that although it can return a state description of the dead state, it cannot pinpoint the conflicting constraints. The problem is exacerbated by the fact that any number of constraints can contribute to an unsatisfiable specification. A greedy algorithm using satisfiability on the constraints such that the conflicting requirements can be found, should be developed for a more complete specification debugging tool environment.

Acknowledgements

The authors will like to thank K. McMillan for help with Cadence SMV, S. Berezin for help with CMU SMV, H. Kapadia for answering PCI questions,

and our colleagues in the Gigascale Silicon Research Center (GSRC) for useful feedback. This research was supported by GSRC contract SA2206-23106PG-2.

References

1. M. Kaufmann, A. Martin, and C. Pixley. “Design Constraints in Symbolic Model Checking” in *International Conference on Computer-Aided Verification*, 1998.
2. PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.2*, December 18 1995.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. “Sequential circuit verification using symbolic model checking” in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
4. K. McMillan. <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>.
5. P. Chauhan, E. M. Clarke, Y. Lu and D. Wang. “Verifying IP-Core based System-On-Chip Designs” in *Proceedings of the IEEE ASIC conference*, September 1999.
6. E.M. Clarke and E.A. Emerson. “Synthesis of synchronization skeletons for branching time temporal logic” in *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981 Lecture Notes in Computer Science*, vol. 131, Springer-Verlag. 1981.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic” in *ACM Transactions on Programming Languages and Systems* 8(2):244-263, April, 1986.
8. A. Mokkedem, R. Hosabettu, and G. Gopalakrishnan. “Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem” in *Proceedings of the Second International Conference, Formal Methods in Computer-Aided Design*, 1998. Lecture Notes in Computer Science, vol. 1522, Springer-Verlag.
9. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*, MIT Press, 1989.
10. G.S. Govindaraju and D.L. Dill. “Counterexample-guided Choice of Projections in Approximate Symbolic Model Checking” in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, 2000. (under review).
11. R.E. Bryant, P. Chauhan, E.M. Clarke, and A. Goel. “A Theory of Consistency for Modular Synchronous Systems” in *Proceedings of the Third International Conference, Formal Methods in Computer-Aided Design*, 2000. (under review).
12. E.M. Clarke, Y. Lu, H. Veith, D. Wang, and S. German. “Executable Protocol Specification in ESL” in *Proceedings of the Third International Conference, Formal Methods in Computer-Aided Design*, 2000. (under review).

SAT-Based Image Computation with Application in Reachability Analysis

Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta

NEC USA CCRL

4 Independence Way, Princeton, NJ 08540

{agupta, jyang, ashar, anubhav}@ccrl.nj.nec.com

Abstract. Image computation finds wide application in VLSI CAD, such as state reachability analysis in formal verification and synthesis, combinational verification, combinational and sequential test. Existing BDD-based symbolic algorithms for image computation are limited by memory resources in practice, while SAT-based algorithms that can obtain the image by enumerating satisfying assignments to a CNF representation of the Boolean relation are potentially limited by time resources. We propose new algorithms that combine BDDs and SAT in order to exploit their complementary benefits, and to offer a mechanism for trading off space vs. time. In particular, (1) our integrated algorithm uses BDDs to represent the input and image sets, and a CNF formula to represent the Boolean relation, (2) a fundamental enhancement called BDD Bounding is used whereby the SAT solver uses the BDDs for the input set and the dynamically changing image set to prune the search space of all solutions, (3) BDDs are used to compute all solutions below intermediate points in the SAT decision tree, (4) a fine-grained variable quantification schedule is used for each BDD subproblem, based on the CNF representation of the Boolean relation. These enhancements coupled with more engineering heuristics lead to an overall algorithm that can potentially handle larger problems. This is supported by our preliminary results on exact reachability analysis of ISCAS benchmark circuits.

1 Introduction

Image and pre-image computation play a central role in symbolic state space traversal, which is at the core of a number of applications in VLSI CAD like verification, synthesis, and testing. The emphasis in this paper is on reachability analysis for sequential system verification. For simplicity of exposition, we focus only on image computation; the description can be easily extended to pre-image computation as well.

1.1 BDD-Based Methods

Verification techniques based on symbolic state space traversal [1, 2] rely on efficient algorithms based on BDDs [3] for computing the image of an input set over a Boolean relation. The input set in this case is the set of present states P , and the Boolean relation is the transition relation T , i.e. the set of valid

present-state, next-state combinations. (For hardware, it is convenient to also include the primary inputs in the definition of T). The use of BDDs to represent the characteristic function of the relation, the input, and the image set, allows image computation to be performed efficiently through Boolean operations and variable quantification. As an example of its application, the set of reachable states can be computed by starting from a set P which denotes the set of initial states of a system, and using image computation iteratively, until a fixpoint is reached.

A number of researchers have proposed the use of *partitioned* transitioned relations [1, 2], where the BDD for the entire transition relation is not built a priori. Typically, the partitions are represented using multiple BDDs, and their conjunction is interleaved with early variable quantification during image computation. Many heuristics have been proposed to find a good quantification schedule, i.e. an ordering of the conjunctions which minimizes the number of peak variables [3, 4]. There has also been an interest in using disjunctive partitions of the transition relations and state sets [5, 6, 7], which effectively splits the image computation into smaller subproblems.

The BDD-based approaches work well when it is possible to represent the sets of states and the transition relation (as a whole, or in a usefully partitioned form) using BDDs. Unfortunately, BDD size is very sensitive to the number of variables, variable ordering, and the nature of the logic expressions being represented. In spite of a large body of work, the purely BDD-based approach has been unreliable for designs of realistic size and functionality.

1.2 Combining BDDs with SAT-Based Methods

An alternative, used extensively in testing applications [8], is to represent the transition relation in Conjunctive Normal Form (CNF) and use Boolean Satisfiability Checking (SAT) for various kinds of analysis. SAT solver technology has improved significantly in recent years with a number of sophisticated packages now available, e.g. [9].

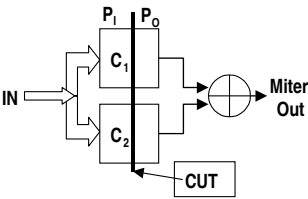


Fig. 1. Miter Circuit for Combinational Verification

For checking equivalence of two given combinational circuits C_1 and C_2 , a typical approach is to prove that the XOR of their corresponding outputs, called the miter circuit output, can never evaluate to 1, as shown in Figure 1. This proof can be provided either by building a BDD for the miter, or by using a

SAT solver to prove that no satisfying assignment exists for the miter output. In cases where the two methods fail individually, BDDs and SAT can also be combined, for example, in the manner shown in Figure 1. A cut is identified in the miter circuit to divide the circuit into two parts: the part P_I of the circuit between the circuit inputs and the cut, and the part P_O of the circuit between the cut and the output. A BDD is built for P_O , while P_I is represented in CNF. A SAT solver then tries to enumerate all valid combinations at the cut using the CNF for P_I , while checking that it is not contained in the on-set of the BDD for P_O [14]. Enumerating the valid combinations at the cut corresponds exactly to computing the image of the input set over the Boolean relation corresponding to P_I . Other ways of combining BDDs and SAT for equivalence checking have also been proposed [15].

For property checking, the effectiveness of SAT solvers for finding bugs has also been demonstrated in the context of bounded model checking and symbolic reachability analysis [16, 17]. The common theme is to convert the problem of interest into a SAT problem, by devising the appropriate propositional Boolean formula, and to utilize other non-canonical representations of state sets. However, they all exploit the known ability of SAT solvers to find a single satisfying solution when it exists. To our knowledge, no attempt has been made to formulate the problems in a way that a SAT solver is used to find *all* satisfying solutions.

In our approach to image computation, we use BDDs to represent state sets, and a CNF formula to represent the transition relation. All valid next state combinations are enumerated using a backtracking search algorithm for SAT that exhaustively visits the entire space of primary input, present state and next state variables. However, rather than using SAT to enumerate each solution all the way down to a leaf, we invoke BDD-based image computation at intermediate points within the SAT decision procedure, which effectively obtains all solutions below that point in the search tree. In a sense, our approach can be regarded as SAT providing a disjunctive decomposition of the image computation into many subproblems, each of which is handled in the standard way using BDDs. In this respect, our work is closest to that of Moon et al. [18], who independently formulated a decomposition paradigm similar to ours. However, there are significant differences in the details, and we defer that discussion to Section 4.

We start by providing the necessary background on a typical SAT decision procedure in the next section. Our proposed algorithm for image computation is described in detail in the sections that follow. Towards the end, we provide experimental results for reachability analysis, which validate the individual ideas and the overall approach proposed by us, and describe some of our work in progress.

2 Background: Satisfiability Checking (SAT)

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem with many applications in computer-aided design, such as test

generation, logic verification and timing analysis. Given a Boolean formula, the objective is to either find an assignment of 0-1 values to the variables so that the formula evaluates to true, or establish that such an assignment does not exist. The Boolean formula is typically expressed in Conjunctive Normal Form (CNF), also called product-of-sums form. Each sum term (clause) in the CNF is a sum of single literals, where a literal is a variable or its negation. An n -clause is a clause with n literals. For example, $(v_i + v_j' + v_k)$ is a 3-clause. In order for the entire formula to evaluate to 1, each clause must be satisfied, i.e., evaluate to 1.

```

Initialize(); // all var made "free"
do {
    Implications();
    status = Bound();
    if (status == contradiction)
        if (active_var->assigned_val ==
            active_var->first_val)
            active_var->assigned_val =
                !active_var->first_val;
        else
            prev_var = Backtrack();
            if (prev_var == NULL)
                soln = no_soln;
                return;
            endif;
        endif
    else
        active_var = Next_free_var();
        if (active_var == NULL)
            soln = found;
            return;
        endif;
        active_var->first_val =
            active_var->assigned_val = Val();
        endif;
    } While ();

```

Fig. 2. Backtracking Search Procedure for SAT

The complexity of this problem is known to be NP-Complete. In practice, most of the current SAT solvers are based on the Davis-Putnam algorithm [11]. The basic algorithm begins from an empty assignment, and proceeds by assigning a 0 or 1 value to one free variable at a time. After each assignment, the algorithm determines the direct and transitive implications of that assignment on other variables, typically called *bounding*. If no contradiction is detected during the implication procedure, the algorithm picks the next free variable, and repeats the procedure. Otherwise, the algorithm attempts a new partial assignment by complementing the most recently assigned variable for which only one value has been tried so far. This step is called *backtracking*. The algorithm terminates either when all clauses have been satisfied and a solution has been found, or when all possible assignments have been exhausted. The algorithm is complete in that it will find a solution if it exists.

Pseudo code for the basic Davis-Putnam search procedure is shown in Figure 2. The function and variable names have obvious meanings. This procedure has been refined over the years by means of enhancements to the `Implications()`, `Bound()`, `Backtrack()`, `Next_free_var()` and `Val()` functions. The GRASP work [12] proposed the use of non-chronological backtracking by performing a conflict analysis, and addition of conflict clauses to the database in order to avoid repeating the same contradiction in the future.

3 Image Computation

The main contribution in our paper is the novel algorithm for image computation by combining BDD- and SAT-based techniques in a single integrated framework. In relationship to current SAT solvers, our contributions are largely specific to their use for image computation. They are orthogonal to the most advanced features found in state-of-the-art SAT algorithms like GRASP [12], and indeed add to them.

3.1 Representation Framework

Our representation framework consists of using BDDs to represent the input and image sets, and a CNF formula to represent the Boolean relation. This choice is motivated by the fact that BDD-based methods frequently fail because of their inability to effectively manipulate the BDD(s) for the transition relation, in its entirety or in partitioned form. Furthermore, since BDDs for the input and image sets might also become large for complex systems, we do not require that a single BDD be used to represent these sets. Any disjunctively decomposed set of BDDs will work with our approach. This setup is shown pictorially in Figure 3. For our current prototype, we use a simple “chronological” disjunctive partitioning, such that whenever the BDD size for a set being accumulated crosses a threshold, a new BDD is created for storing future additions to the set. We are exploring use of alternative representations to manage these sets.

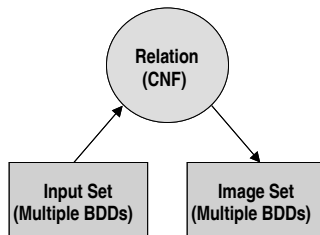


Fig. 3. The CNF-BDDs Representation Setup

3.2 Image Computation Using CNF-BDDs

The standard image computation formula is shown below in Equation (1), where x , y , and w denote the set of present state, next state, and primary input variables, respectively; $P(x)$ denotes the input set, and $T(x, w, y)$ denotes the transition relation.

$$Image(P, T)(y) = \exists x, w. P(x) \wedge T(x, w, y) \quad (1)$$

In our framework, $P(x)$ and $Image(y)$ are represented as (multiple) BDDs, while T is represented as a CNF formula in terms of x , w , y and some additional variables z , which denote internal signals in the circuit. We compute the image set by enumerating all solutions to the CNF formula T , and recording only the combinations of y variables, while restricting the values of x variables to those that satisfy $P(x)$. Note that by restricting the x variables to satisfy $P(x)$, we are effectively performing the conjunction in the above formula. This restriction is performed by what we call *BDD Bounding*. Essentially, during the SAT search procedure, any partial assignment to the x variables that does not belong to the on-set of the BDD(s) $P(x)$ is pruned immediately [10]. Note also, that by enumerating *all* (not a single) solution to the CNF formula, and by considering combinations of only y variables among these solutions, we are effectively performing a quantification over all the other variables (x , w , z).

We also use $Unreached(y)$ as a care-set for the image set. In applications such as reachability analysis where image computation is performed iteratively, this set can be computed as the negation of the current set of reached states. Again, by using BDD(s) to represent $Unreached(y)$, we can obtain additional pruning of the SAT search space by performing BDD Bounding against this image care-set. To summarize, we use the following equation for image computation:

$$Image(P, T)(y) = \exists x, w, z. P(x) \wedge T(x, w, z, y) \wedge Unreached(y) \quad (2)$$

4 BDD Bounding

A naive approach for performing BDD Bounding is to enumerate each complete SAT solution up to the leaf of the search tree, and then check if the solution satisfies the given BDD(s). This is obviously inefficient since the number of SAT solutions may be very large.

In our setup, the x/y variables are shared between the input/image set BDD(s) and the CNF formula. Therefore, whenever a value is set to or implied on one of these variables in SAT, we can check if the intersection of the partial assignment with the given BDD(s) is non-null. If it is indeed non-null, the SAT procedure can proceed forward. Otherwise it must backtrack, since no solution consistent with the conjunctions can be found under this subtree. In our earlier work on combinational verification, we had called this the *Early Bounding* approach [10], and had demonstrated a significant reduction in the number of backtracks due to pruning off large subspaces of the search tree. Note that the smaller the bounding set, the greater the pruning, and the faster the SAT solver is likely to be.

```

Bound(sat,lit,input_bdd) {
  if (lit_is_input_set_variable(lit)){
    new_bdd =
    project_variable_in_bdd(input_bdd,lit);
    if (bdd_equal_zero(new_bdd))
      return contradiction;
    else
      input_bdd = new_bdd;
  } // rest of the procedure is unchanged
}

```

Fig. 4. Pseudo-code for BDD Bounding

4.1 Bounding against the Image Set: A Positive Feedback Effect

In addition to bounding against the input set $P(x)$ and the image care-set $Unreached(y)$, a fundamental speed-up in our image computation procedure can be obtained by also bounding against the BDDs of the currently computed image set denoted $Current(y)$. Note that $Unreached(y)$ does not change during a single image computation, while $Current(y)$ is updated dynamically, as new solutions for the image set are enumerated. Therefore, if a partial assignment over y variables is contained in $Current(y)$, it implies that any extension to a full assignment has already been enumerated. Therefore, it serves no purpose for the SAT solver to explore further, and it can backtrack. As a result, a positive feedback effect is created in which the larger the image set grows, the faster the SAT solver is likely to be able to go through the remaining portion of the search space.

4.2 Implementation Details: Bounding the x Variables

For BDD Bounding against $P(x)$, we modify the `Bound()` function of Figure 2, so that it checks the satisfaction of a partial assignment on x variables with the on-set of the BDDs for $P(x)$. Again, if the partial assignment has a null intersection with each BDD, the SAT solver is made to backtrack, just as if there were a contradiction.

The pseudo-code for the `Bound()` procedure for a single BDD is shown in Figure 1. In this procedure, the initial argument is the BDD for $P(x)$, and the recursive argument maintains its projected version down the search tree. The `project_variable_in_bdd()` procedure, can be easily implemented by using either the `bdd_substitute()` or the `bdd_cofactor()` operations available in standard BDD packages [21]. Unfortunately, these standard BDD operations represent a considerable overhead in terms of unique table and cache lookups. In our implementation, we use a simple association list to keep track of the x variable assignments. Projecting (un-projecting) a variable is accomplished simply by setting (un-setting) its value in the association – a constant-time operation. However, checking against a `zero_bdd` requires a traversal of the argument BDD, by taking branches dictated by the variable association. If any path to a `one_bdd` is found, the traversal is terminated, and the BDD is certified to be not equal to the `zero_bdd`. In the worst case, this takes time proportional to the size of

the BDD. As a further enhancement, for each BDD node, we associate a value indicating the presence or absence of a path to a `one_bdd` from that node. This bit must be modified only if the value of a variable below this node is modified. As a result, the average complexity of the emptiness check is likely better than the BDD size.

4.3 Implementation Details: Bounding the y Variables

During the SAT search, if the intersection of a partial assignment over y variables and $(Unreached(y) \wedge !Current(y))$ is non-null, exploration should proceed further down that path, otherwise it can backtrack.

```

/* Each BDD node has a user-defined field that indicates if the sub-graph
below it is tautologically one, zero or neither (TWO_BDD) */
/* B_array is an array of BDD nodes, V_array is the array of variables and
their values */
bdd_equal_zero(B_array, V_array) {
  /* Leaves */
  if (any BDD in B_array is ZERO_BDD)
    return ZERO_BDD;
  if (each BDD in B_array is ONE_BDD)
    return ONE_BDD;
  if (only one BDD in B_array is TWO_BDD)
    return TWO_BDD;

  /* Non-leaf case */
  NewB_array = remove_one_bdds(B_array);
  if (is_in_cache(NewB_array, &Val))
    return Val;

  /* get top var from all the BDDs in NewB_array */
  Top_var = get_top_var(NewB_array);
  ThenB_array = get_then(NewB_array, Top_var);
  ElseB_array = get_else(NewB_array, Top_var);
  Proj_value = get_proj_val(Top_var, V_array);
  if (Proj_value == ONE) {
    Val = bdd_equal_zero(ThenB_array, V_array);
  } else if (Proj_value == ZERO) {
    Val = bdd_equal_zero(ElseB_array, V_array);
  } else {
    Val0 = bdd_equal_zero(ThenB_array, V_array);
    Val1 = bdd_equal_zero(ElseB_array, V_array);
    if (Val0 == Val1) {
      Val = Val0;
    } else {
      Val = TWO_BDD;
    }
  }
  insert_in_cache(newB_array, Val);
  return Val;
}

```

Fig. 5. Determining Emptiness of the Product of Multiple BDDs with Projections

Recall that we allow use of a disjunctive partitioning of the reached state set $R = \cup_i R_i$. Therefore, both the $Unreached$ and $!Current$ sets can be represented as product of BDDs, i.e. $Unreached(y) = \cap_i !R_i(y)$, and $!Current(y) = \cap_i !Current_i(y)$. Rather than performing an explicit product of the multiple BDDs, the partial assignment over y variables is projected separately onto each

BDD. Then the the multiple BDDs are traversed in a lock-step manner by using a modified `bdd_equal_zero()` procedure, to determine if there exists a path in their product that leads to a `one_bdd`. The pseudo code for this is shown in Figure 1, where the given procedure assumes that projection of variable values onto the individual BDDs has already been carried out. In the actual implementation, the projection of variables and detection of emptiness are done in a single pass, along with handling of complemented BDD nodes. The worst-case complexity is that of actually computing a complete product, but in practice the procedure terminates as soon as any path to `one_bdd` is found.

5 BDDs at SAT Leaves

So far we have explained our algorithm for image computation in terms of enumerating all solutions of the CNF formula using SAT-solving techniques, while performing BDD Bounding where possible in order to prune the search space. This still suffers from some drawbacks of a purely SAT-based approach, i.e. solutions are enumerated one-at-a-time, without any reuse. To some extent this drawback is countered by examining partial solutions (cubes) for inclusion and for pruning, but we can actually do better.

It is useful in this regard to compare a purely SAT-based approach vs. a purely BDD-based approach. In essence, both work on the same search space of Boolean variables – SAT solvers use an explicit decision tree, while BDD operations work on the underlying DAGs. A BDD-based approach is more suitable for capturing all solutions simultaneously. However, due to the variable ordering restriction, it can suffer from a size blowup in the intermediate/final results. On the other hand, a SAT decision tree has no variable ordering restriction, and can therefore potentially manage larger problems. However, since it is not canonical, many subproblem computations may get repeated.

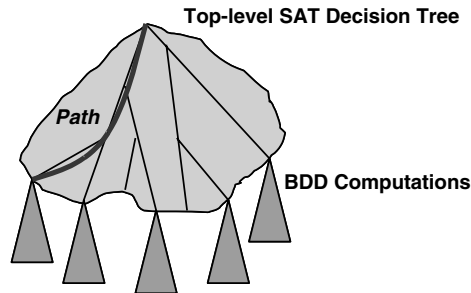


Fig. 6. BDDs at SAT Leaves

In order to combine the relative advantages of both, we use a SAT decision tree to organize the top-level search space. Within this tree, along any path, rather than using the SAT-solver to explore the tree further, we can invoke a

BDD-based approach to compute all solutions in the sub-tree under that path. This integrated scheme, which we call *BDDs at SAT Leaves*, is illustrated pictorially in Figure 2. In a sense, the SAT decision tree can be regarded as a disjunctive partitioning of a large problem at the root into smaller subproblems at the leaves, each of which can be handled by a purely BDD-based approach.

5.1 Leaf Subproblem: BDD-Based Image Computation

The formulation of the BDD subproblem to be solved at each leaf of the SAT decision tree is shown below:

$$New(y) = Path(y') \wedge \exists x'', w'', z''. P(x)|_{Path(x')} \wedge Unsat(x'', w'', z'', y'') \quad (3)$$

This computes the image set solutions New from a sub-tree rooted at the end of a path in the SAT decision tree. Here, for a set of variables v , the assigned set is denoted v' , and the unassigned set is denoted v'' . $Path(x')/Path(y')$ denote the BDDs representing the partial assignment of x/y variables along the path. $Unsat(x'', w'', z'', y'')$ denotes the product of all unsatisfied clauses at the end of the path, projected by the assigned variables along that path, expressed in terms of the unassigned variables appearing in the original CNF formula. Finally, $P(x)|_{Path(x')}$ denotes the restriction of the set $P(x)$ to the partial assignments of x along the path.

Note that in this equation, the part following the existential quantification is identical in formulation to a standard purely BDD-based approach. The difference is only in the *granularity* of the Boolean relation $Unsat$, and its conjunctive decomposition. In a standard approach, the Boolean relation $T(x, w, y)$ is a transition relation, expressed in terms of the present state, primary input, and next state variables only. Furthermore, its conjunctive decomposition is typically based on splitting the next state variables.

In our approach, the Boolean relation is expressed as a CNF formula over the set of present state, primary input, next state, and intermediate variables denoting signals on internal gates that implement the next state logic of the sequential circuit. Furthermore, the conjunctive decomposition follows the structural decomposition of the circuit into gates. Though this finer-grained approach must handle more number of Boolean variables than the standard approach, it also allows a greater potential for early quantification, which has been noted to help overcome the blowup during image computation (described in detail in the next section).

Another benefit of using the fine-grained CNF partitions is that there is no penalty for performing pre-image computations. Many researchers have noted that backward symbolic traversal is less efficient than forward traversal. This is partly due to having to handle the typically irregular unreachable part of the state space. Furthermore, most methods use partitions based on splitting the next state (y) variables, while sharing the present state (x) variables. This scheme is good for performing image computations with early quantification of x variables, but it does not work very well for pre-image computations where the y variables need to be quantified. In contrast, our fine-grained CNF formulation

is symmetric with respect to the x and y variables. Therefore, our method can be applied equally well for image as well as pre-image computations.

5.2 Leaf Subproblem: Quantification Schedule

In practice, it is important to choose a good quantification schedule, i.e. an ordering on the conjunctions of the partitions that avoids intermediate blowup during image computation. Typically, a good schedule tries to minimize the number of active variables in a linearized schedule, by analyzing the variable support sets of the individual partitions [14, 15, 16].

```

Leaf_Image_Computation() {
  B = {projected P(x), projected unsat_clauses};    // set of BDDs
  do {
    v = min_cost_variable(B);    // choose variable v
    C = {b | b ∈ B, b depends on v};    // gather conjuncts for v
    c = and_smooth(C, v);    // quantify v along with conjunction
    replace(B, C, c);    // replace conjuncts C in B by c
  } while (variables_to_be_quantified);
  c = and(B);
  new = and(path(y), c);
}

```

Fig. 7. Leaf Image Computation

For each leaf image computation, the pseudo-code for the quantification schedule is shown in Figure 7. We start with a collection B of BDDs consisting of the projected $P(x)$ (and potentially $Unreached(y)$), and a BDD for every projected unsatisfied clause. Next, we heuristically select a variable v to be quantified. We greedily choose the minimum cost variable, where cost is estimated as the product of the individual BDD sizes that the variable appears in. Once v is selected, we gather in set C all conjuncts that v appears in. This is followed by conjunction and quantification of v in C , and this result replaces the set C in B . (Since the y variables cannot be quantified, we never choose them.) This basic loop is iterated until no more variables can be quantified. The remaining BDDs (with only y variables) are conjoined together, and the result is conjoined with $path(y)$ (the cube of assigned y variables), to give the set of *new* image solutions corresponding to that path.

Note that this formulation does not depend on a live variable analysis over a linearized schedule but considers the actual BDD sizes for selection. Therefore, it is better able to balance the computation in the form of a tree of conjunctions, rather than a linear series of conjunctions. In our experiments, this heuristic performed far better than others based on variable supports.

```

1  /* This function finds all solutions to a SAT Clause database,
2  using GRASP with BDD bounding, BDDs-at-SAT Leaves */
3  Find_all_solutions(Clauses database, Bdds) {
4    if (Preprocess() == FAILURE)
5      return FAILURE;
6    if (Bdd_bounding() == FAILURE)
7      return FAILURE;
8    while (1) { // loop #1: exploring dec. tree to incr. depth
9      if (BDDs_at_Leaves()) {
10         handle_bdd_solution(); // stop exploring tree
11         /* for other solutions: backtrack chronologically */
12         if (Backtrack_chrono() == FAILURE) {
13           return NO_MORE_SOLUTIONS;
14         } // else backtracking takes place */
15       } else {
16         /* explore tree to increased depth */
17         if (Select_next_variable() == FAILURE)
18           break; // out of loop #1 */
19       }
20     } // loop #2: check conflicts after decision / backtracking
21     /*
22     while (1) {
23       /* loop #3: while there is a logical conflict */
24       while (Deduce() == CONFLICT) {
25         if (First_val() || Implied_Val()) {
26           /* perform diagnosis */
27           if (Diagnose() == CONFLICT) {
28             /* the conflict cannot be resolved any more */
29             return NO_MORE_SOLUTIONS;
30           }
31         }
32       }
33     }
34   } // else conflict driven backtracking takes place */
35   } else {
36     /* second value of var: backtrack chronologically */
37     if (Backtrack_chrono() == FAILURE)
38       return NO_MORE_SOLUTIONS;
39   } // else backtracking takes place */
40 } // end of loop #3 */
41 /* at this point, there is no conflict */
42 if (Bdd_Bounding() == SUCCESS) {
43   if (Solution_found()) {
44     handle_sat_solution();
45     /* for other solutions: backtrack chronologically */
46     if (Backtrack_chrono() == FAILURE)
47       return NO_MORE_SOLUTIONS;
48   } else { // else backtracking takes place
49     /* unresolved clauses: increase depth */
50     break; // out of loop #2
51   }
52 } // BDD bndng fails: backtrack chronologically
53 if (Backtrack_chrono() == FAILURE)
54   return NO_MORE_SOLUTIONS;
55 } // end of loop #2
56 } // end of loop #1
57 return SUCCESS;
58 }
59 }
60 }
61 }

```

Fig. 8. Complete Image Computation Procedure

6 The Complete Image Computation Procedure

Our complete procedure for enumerating all solutions of the image set is shown in Figure 8. It is based on the publicly available GRASP SAT-solver [14]. We start by describing its original skeleton. After the initial preprocessing, the procedure consists of an outer loop #1 (line 10) that explores the SAT decision tree to increased depth if necessary. The inner loop #2 (line 24) is used primarily to propagate constraints and check for conflicts after either a decision variable is chosen, or after backtracking takes place to imply a certain value on a variable. Loop #3 (line 26) actually performs the deduction to check for contradictions and tries to resolve the conflict using diagnosis until there is no more conflict. In GRASP, clauses are added to record causes of all backtracking operations, including those used to enumerate multiple solutions.

The completeness argument for our procedure with respect to finding all solutions of the image set is based largely on the completeness of the original procedure in GRASP [14]. The additions we have made to the original procedure consist of introducing the techniques of *BDDs at SAT Leaves* (lines 11-18), and *BDD Bounding* (lines 7-8, lines 42-57). The only other modification we have made is to perform conflict analysis only if the value of the decision variable is the first value being tried, or if its second value has been implied (line 27).

The correctness of finding all BDD solutions at the leaves, and of pruning the search space when BDD Bounding fails follows from the arguments described in the previous sections. Note that in both these cases, we perform a chronological

backtracking (lines 14-17, lines 54-56) in order to search for the next solution. In case BDD Bounding succeeds (line 42), we check whether a solution is found, i.e. whether all clauses are satisfied. If they are, a SAT solution has been found, which is handled in the usual way, followed by chronological backtracking to find the next solution (lines 45-48).

The reason for the modification (line 27) is that we do not wish to add clauses to record the causes of chronological backtracking. In our modified GRASP algorithm, chronological backtracking takes place after a solution has been found, or after BDD Bounding fails. However, when clauses for chronological backtracking are not recorded, GRASP's conflict analysis during diagnosis becomes incomplete, and it may be erroneous to perform non-chronological backtracking based on this conflict analysis. Therefore, if the second value of a variable is implied by some clause (either an original, or a conflict clause), we do allow diagnosis to take place. Otherwise, we disable non-chronological backtracking by not performing any diagnosis at all. Instead, we perform a simple chronological backtracking (lines 34-39). Note that performing chronological backtracking instead of non-chronological backtracking can at most affect the procedure's efficiency, not its completeness.

7 Why SAT?

As mentioned earlier, there has been recent interest in using disjunctive decompositions of the image computation problem using purely BDDs, with substantially improved practical results [1, 2]. Our use of a SAT decision tree to split the search tree, and use of the BDD-based image computations at its leaves to perform the conjoining, results in a similar decomposition. However, in our view, SAT provides many more advantages than just a disjunctive decomposition, which also differentiate our approach from the rest.

In particular, it allows us to easily perform implications of a variable decision (splitting). In principle, deriving implications can be done in non-SAT contexts as well, e.g. directly on circuit structure, using BDDs etc. However, to our best knowledge, this has not been done in practice for image computation. By using a standard state-of-the-art SAT package [3], we are utilizing the years of progress in this direction, as well as in related techniques of efficient backtracking and conflict analysis, which all help toward pruning the underlying search space. Our use of BDD Bounding is an additional pruning technique, which allows us to perform early backtracking without even invoking a BDD-based leaf computation.

Another difference of our approach from the rest is in the granularity of our underlying search space. Since we focus on the CNF formula for the transition relation, which is derived directly from a gate-level structural description of the design, we obtain a very fine-grained partition of the relation, which is also symmetric with respect to image and pre-image computations. This allows us to split the overall into much finer partitions, where decision (splitting) variables can also be internal signals. We use both BDD-based and SAT-based criteria for selection of these variables, e.g. estimate of cofactor BDD sizes [4], number

of clauses a variable appears in, etc. We are also exploring SAT-based criteria targeted towards finding multiple, and not single, solutions. For each partition itself, the finer level of granularity allows us to exploit the benefits of early quantification to a greater degree. This is reflected in our BDD-based quantification schedule algorithm, which uses different criteria (actual BDD sizes) for selecting the variable to be quantified, and is organized as a tree of conjunctions, rather than a linear series.

Finally, our aim is to combine SAT and BDDs in a seamless manner in order to facilitate a smooth and adaptive tradeoff between time and space for solving the image computation problem. In our algorithm, the move from SAT to BDDs occurs when a BDD subproblem is triggered. Ideally, we would like to do this whenever we could be sure that the BDDs would not blow up. However, there seems to be no simple measure to predict this *a priori*. We are currently experimenting with several heuristics based on number of unassigned variables, size of the projected $P(x)$ set etc. We have also implemented a simple timeout mechanism for the BDD subproblem, which allows us to return back to SAT, in order to perform some more splits (unlike [14]). Since CNF formulas and BDDs are entirely interchangeable, the boundary between SAT and BDDs is somewhat arbitrary. In principle, it is possible to freely intermix CNFs and BDDs for various parts of the circuit, and perform required analysis on the more appropriate representation. Our approach is a step in this direction.

8 Experiments

We have implemented an initial prototype of our image computation algorithm based on the CUDD BDD package [22] and the GRASP SAT solver [23]. This section describes our experimental results on some ISCAS benchmark circuits known to be difficult for reachability analysis. All experiments were run on an UltraSPARC workstation, with a 296 MHz processor, and 768 MB memory.

Since our main contribution here is to make the core step of image computation more robust, we only focus on experiments for exact reachability analysis. Our algorithm can be easily adapted and enhanced in many orthogonal directions such as its use in approximate reachability analysis, invariant checking, and model checking. We are currently working on porting this prototype to VIS [4] in order to use its infrastructure for such applications, and also to have access to a wider set of benchmarks.

A comparison of our prototype, which we call the CNF-BDD prototype, with VIS [4] is shown in Table 1. It shows results for performing an exact reachability analysis using pure breadth-first traversal on some benchmark circuits known to be difficult to handle in practice. The circuit name and number of latches are shown in Columns 1 and 2, respectively. For our approach, a measure of circuit complexity is the number of variables appearing in our CNF representation of the transition relation – shown in Column 3. The number of steps completed is shown in Column 4, where a “(p)” indicates partial traversal. Column 5 reports the number of states reached. In Column 6, we report the CPU time taken (in seconds) by VIS. For these experiments we used a timeout of 10 hours. However,

Ckt	#FF	#Vars	#Steps	#Reached States	Vis Time (s)	CNF BDD				
						Time (s)	Mem (MB)	Peak (M Nodes)	Leaves	Bounds
s1269	37	624	10	1.13E+09	3374	1877	46	0.6	15150	5278
s1512	57	866	1024	1.66E+12	2362	6337	28	0.34	13289	3565
s1423	74	748	11 (p)	7.99E+09	7402	2425	40	1.16	217	329
			13 (p)	7.96E+10	--	5883	274	6.86	330	452
s5378	164	2978	6 (p)	2.47E+16	31346	19024	197	2.4	668	348
			8 (p)	2.36E+17	--	29230	202	2.4	981	421

Table 1. Results for Exact Reachability Analysis

sometimes VIS runs into memory limitations before the timeout itself, indicated as “--” in this column. Columns 7 through 11 provide numbers for our CNF-BDD prototype. Column 7 reports the CPU time taken (in seconds). Columns 8 and 9 report the memory used, and the number of peak BDD nodes, as reported by the CUDD package. To give an idea of the efficiency of our modified SAT solver, we report the number of BDDs subproblems (Leaves), and the number of backtracks due to BDD Bounding (Bounds) in Columns 10 and 11, respectively. In all our experiments, we did not observe any non-chronological backtracking in the SAT solver. Therefore, the total number of backtracks during image computation for all steps is the sum of Columns 10 and 11.

It can be seen that the performance of our CNF-BDD prototype is better than VIS in 3 of 4 circuits. For s1512, our prototype is worse likely due to the large number of steps – 1024, and the overhead in every step of re-starting the SAT solver. For s1269, our prototype performs better than VIS by about a factor of 2. For both of these circuits, the CNF-BDD numbers are worse than those reported recently by Moon et. al [17] – 891 sec. for s1269, and 2016 sec. for s1512. The real gains from our approach can be seen in the more difficult circuits s1423 and s5378, where neither VIS nor our prototype can perform complete traversal. For s1423, VIS was able to complete up to step 11. For the same number of steps, our prototype is faster than VIS by more than a factor of 3. In fact, our prototype is able to complete two additional steps of the reachability computation in the time allotted compared to VIS. Similarly, for s5378, our prototype is faster than VIS up to step 6, and is able to complete 2 additional steps in the time allotted. As can be seen from the Columns showing Mem and Peak, our approach seems not to be memory bound yet, for these experiments. On the other hand, VIS gets memory bound in the allotted time. Furthermore, the number of backtracks is also well under control, with BDD Bounding being very effective in pruning the SAT search space. For all our experiments, the overhead of computing the BDD Bounding information was negligible.

To contrast the memory requirements of our approach with a standard purely BDD-based approach, we conducted detailed experiments for s1423. At this time, we were not able to change the VIS interface in order to extract the memory usage statistics we needed. Therefore, these experiments were conducted using a

s1423	Nano Trav				CNF-BDD				
Step	#Reached	Time(s)	Peak	Live	Time(s)	Peak	Live	Leaves	BoundB
1	545	2	14308	2627	0.7	22484	249	2	12
2	3345	2.5	19418	3847	4.9	31682	337	8	14
3	55569	5.6	32704	4010	4.1	42924	550	9	18
4	3.92E+05	14.1	35770	6766	5.6	50078	1352	13	22
5	2.08E+06	39.4	55188	15773	8.4	50078	2629	9	19
6	8.49E+06	81.4	74606	19824	11.6	64386	6562	12	20
7	3.37E+07	391.7	17696	35849	20.8	82782	13061	10	19
8	1.11E+08	867.4	320908	64126	27.4	123662	33262	8	17
9	4.90E+08	3050	888118	199237	125.4	224840	93492	34	61
10	1.68E+09	7991.3	1634178	413060	521.8	531440	253527	58	72
11	7.99E+09	18705.2	4027702	650065	1357.1	1168146	585034	51	55
12	2.30E+10	--	--	--	2100.8	4259696	1538532	33	44
13	7.96E+10	--	--	--	1694.3	6865796	4098655	80	79
					5882.9			327	452

Table 2. Results on Memory Usage for Circuit s1423

stand-alone traversal program called “nanotrav”, which is distributed along with the CUDD package [24]. In general, nanotrav performs worse than VIS, since it does not include sophisticated heuristics for early quantification or clustering of the partitioned transition relation. This limitation does affect the memory requirements to some extent, but we present Table 2 mainly to show the overall trend. In this table, the number of steps and reached states is shown in Columns 1 and 2. For nanotrav, Column 3 reports the CPU time taken (in seconds). The number of peak nodes and live nodes at the end of each image computation are shown in Columns 4 and 5, respectively. Columns 6, 7, 8 report the same for our CNF-BDD approach. We also report the number of BDDs at SAT Leaves (Leaves), and the number of backtracks due to BDD Bounding (Bounds) in Columns 9 and 10, respectively.

As can be seen clearly from this table, the memory requirements continue to grow with the number of steps. However, this growth is at a faster pace for nanotrav than it is for the CNF-BDD prototype. At the 11th step, the peak BDD nodes for nanotrav are greater than for CNF-BDD by a factor of 4. Not surprisingly, the CNF-BDD can complete two more reachability steps in about the same time. Furthermore, the CNF-BDD prototype is still not limited by memory while performing the 14th step, but is forced to time out after 10 hours.

In our approach, we can tradeoff between SAT solvers and BDDs by dynamically changing the conditions for triggering BDDs at SAT leaves. However, this only provides a memory-time tradeoff for the purpose of image computation. It does not reduce the memory requirements in using monolithic BDDs for representing state sets. As described in the earlier sections, our approach is

completely suited for handling state sets in the form of disjunctive partitions (multiple BDDs). We are currently working on extending our prototype in this direction.

9 Conclusions and Ongoing Work

In conclusion, we have presented an integrated algorithm for image computation which combines the relative merits of SAT solvers and BDDs, while allowing a dynamic interaction between the two. In addition, the use of a fine-grained CNF formula allowed us to explore the benefits of early quantification in the BDD subproblems. This can potentially find application in purely BDD-based approaches as well.

Apart from extending our prototype to handle other applications such as approximate traversal, invariant checking and CTL model checking, a number of enhancements to the basic image computation strategy are possible. Specifically, we are considering various forms of partitioning including disjunctive partitioning of the reached set, exploiting disjoint partitions in the CNF formula (and maximizing this effect through appropriate variable selection), and partitioning the circuit structurally. The SAT-BDD approach works best when it has a large reached set to bound against. We are experimenting with the use of an under-approximate traversal as a preprocessing step to generate a large reached set, before starting an exact traversal. Similarly, the SAT-BDD framework naturally allows exploration of various combinations of DFS, BFS, and hybrid search techniques targeted at finding bugs.

References

1. P. A. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, 1999.
3. R. K. Brayton et al. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, June 1996.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
5. J. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proceedings of the International Conference on Computer-Aided Design*, pages 570–576, 1998.
6. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th Design Automation Conference*, pages 403–407, June 1991.
7. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, Apr. 1994.

8. G. Cabodi, P. Camurati, and S. Quer. Improving the efficiency of BDD-based operators by means of partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(5):545–556, May 1999.
9. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, June 1989.
10. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–205, 1960.
11. D. Geist and I. Beer. Efficient model checking by automatic ordering of transition relation partitions. In *Proceedings of the International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310, 1994.
12. A. Gupta and P. Ashar. Integrating a Boolean satisfiability checker and BDDs for combinational verification. In *Proceedings of the VLSI Design Conference*, Jan. 1998.
13. T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, Jan. 1992.
14. J. P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD thesis, EECS Department, University of Michigan, May 1995.
15. J. P. Marques-Silva and K. A. Sakallah. Grasp: A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, Nov. 1996.
16. J. P. Marquez-Silva. Grasp package.
<http://algorithms.inesc.pt/~jpms/software.ntml>
17. I.-H. Moon, J. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the Design Automation Conference*, pages 23–28, June 2000.
18. A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. Sangiovanni-Vincentelli. Reachability analysis using partitioned ROBDDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 388–393, 1997.
19. R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *International Workshop for Logic Synthesis*, May 1995. Lake Tahoe, CA.
20. F. Somenzi et al. CUDD: University of Colorado Decision Diagram Package.
<http://vlsi.colorado.edu/~fabio/CUDD/>
21. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, 1990.
22. P. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138, 2000.

SAT-Based Verification without State Space Traversal

Per Bjesse and Koen Claessen

Department of Computing Science
Chalmers University of Technology, 412 96 Göteborg
`{bjesse,koen}@cs.chalmers.se`

Abstract. Binary Decision Diagrams (BDDs) have dominated the area of symbolic model checking for the past decade. Recently, the use of satisfiability (SAT) solvers has emerged as an interesting complement to BDDs. SAT-based methods are capable of coping with some of the systems that BDDs are unable to handle.

The most challenging problem that has to be solved in order to adapt standard symbolic model checking to SAT-solvers is the boolean quantification necessary for traversing the state space. A possible approach to extending the applicability of SAT-based model checkers is therefore to reduce the amount of traversal.

In this paper, we investigate a BDD-based verification algorithm due to van Eijk. Van Eijk's algorithm tries to compute information that is sufficient to prove a given safety property directly. When this is not possible, the computed information can be used to reduce the amount of traversal needed by standard model checking algorithms. We convert van Eijk's algorithm to use a SAT-solver instead of BDDs. We also make a number of improvements to the original algorithm, such as combining it with recently developed variants of induction. The result is a collection of substantially strengthened and complete verification methods that do not require state space traversal.

1 Introduction

Symbolic model checking based on satisfiability (SAT) solvers [1, 2, 3] has recently emerged as an interesting complement to model checking with Binary Decision Diagrams (BDDs) [4]. There are a number of systems which are not suited to be effectively verified using BDD-based model checkers, but can be verified using SAT-based methods. The use of SAT-solvers rather than BDDs also has advantages such as freeing the user from providing good variable orderings, and making the number of variables in the system less of a bottleneck. However, the boolean quantification that is necessary for computing characterisations for sets of predecessors (and successors) of states can sometimes lead to excessively large formulas in SAT adaptations of standard model checking algorithms.

In the hope of alleviating these problems, we investigate a BDD-based algorithm due to van Eijk [1] that attempts to verify safety properties of circuits without performing state-space traversal. The main idea behind the algorithm is to use induction to cheaply compute points in the circuit that always have the same value (or always have opposite values) in the reachable state space. This information sometimes directly implies the safety properties. If such a direct proof is not possible, the computed information can be used to decrease the number of necessary fixpoint iterations in backwards reachability algorithms. Van Eijk [1] has used the algorithm to directly prove equivalence between the original circuits and synthesised and optimised versions of 24 of the 26 circuits in the ISCAS'89 benchmark suite.

We are specifically interested in using van Eijk's algorithm to prove safety properties of circuits that are hard to represent using BDDs. Also, when a direct proof is not possible, we want to use the computed information to reduce the amount of state space traversal in exact SAT-based model checking methods as this could decrease the amount of necessary quantification drastically. As a consequence, we want to find alternatives to the use of BDDs in the original analysis. Van Eijk's algorithm also has the drawback of always computing the *largest* possible set of equivalences, even when this is not needed for the verification of the particular safety property at hand. In some cases this can become too costly; we would therefore like to be able to control how much work we put into finding equivalences.

We solve the two problems by converting the algorithm to use propositional formulas to represent points in the circuit, and by applying Stålmarck's saturation algorithm [2, 3] rather than BDDs for discovering equivalences between points.

The resulting algorithm is generalised in three ways. First, we make the algorithm complete by changing the induction scheme that is used in the method to some recently developed stronger variants of induction [4]. Second, we modify the algorithm to also discover implications between points in the circuit. Third, we demonstrate that van Eijk's algorithm can be viewed as an approximate forwards reachability analysis, and use this insight to construct the dual approximate backwards reachability algorithm and a mutual improvement algorithm.

The information that is computed by the resulting algorithms can in principle be used together with any BDD- or SAT-based model checking method. We show some benchmarks that demonstrate that the methods on their own can be very powerful tools for checking safety properties. For example, we use the algorithms to verify a non-trivial industrial example that previously has been out of reach for the SAT-based model checker FixIt [5].

2 Van Eijk's Method: Finding Equivalence Classes

In this section, we describe van Eijk's method [1]. In the original paper it is presented as a method for equivalence checking of sequential synchronous circuits.

However, while using the method we have observed that it can work well also for general safety property verification.

Basic Idea. The idea behind van Eijk’s algorithm is to find the points in the circuit which have the same value (or have opposite values) in all reachable states. This information can then be used to either directly prove the safety property or to strengthen other verification methods.

The information is represented as an equivalence relation over the points of the circuit and their negations. The algorithm computes such an equivalence relation by means of a fixed point iteration. It starts with the equivalence relation that necessarily holds between the points in the initial state. Then it improves the relation by assuming that the equivalences hold at one time instance and deriving the subset of these equivalences that must hold in the next time instance. After a number of consecutive improvements, a fixed point is reached. The resulting equivalence relation is satisfied by the initial states, and is moreover preserved by any circuit transition. Therefore, it must hold in all reachable states.

Before we give a more precise description of van Eijk’s algorithm, we first introduce some definitions.

Formulas. We describe the systems we are dealing with using propositional logic formulas. These are syntactic objects, built from variables like x and y , boolean values 1 and 0, and connectives \neg , \wedge , \vee , \Rightarrow , and \Leftrightarrow . We say that a formula is *valid* if and only if it evaluates to 1 for all variable assignments under the usual interpretation of the connectives.

State Machines. We represent sequential synchronous circuits as state machines in the standard way [1], where the set of states is the set of boolean valuations of a vector s of variables; one variable for each input and internal latch. As we do not restrict the input part of the states, these state machines are non-deterministic. The standard representation also guarantees that every state has at least one outgoing transition.

We characterise the set of initial states and the transition relation of the state machine by the propositional logic formulas $\text{Init}(s)$ and $\text{Trans}(s, s')$, respectively. In other words, $\text{Init}(s)$ is satisfied exactly by the initial states, and $\text{Trans}(s, s')$ is satisfied precisely when there is a transition between the states s and s' . The safety property of the system we want to verify is represented by the formula $\text{Prop}(s)$.

Example 1. Assume that we want to decide whether the two subcircuits in Figure 1 are equivalent. This amounts to checking whether the signal p is always true. Let us construct the necessary formulas. There are four state variables—one for every input and one for every delay component—so $s = (x, d_1, d_2, d_3)$. Since the delay components have an initial value of 0, the formula for the initial states becomes:

$$\text{Init}(x, d_1, d_2, d_3) = \neg d_1 \wedge \neg d_2 \wedge \neg d_3$$

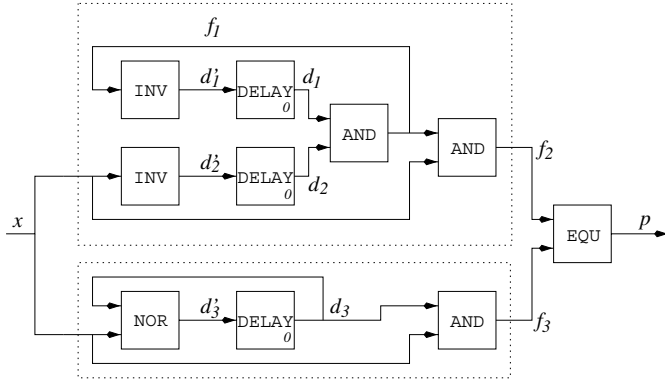


Fig. 1. An example circuit

Looking at the logic contained in the circuit, we can write down the formula for the transition relation:

$$\begin{aligned} \text{Trans}(x, d_1, d_2, d_3, x', d'_1, d'_2, d'_3) = \\ (d'_1 \Leftrightarrow \neg(d_1 \wedge d_2)) \wedge (d'_2 \Leftrightarrow \neg x) \wedge (d'_3 \Leftrightarrow \neg(x \vee d_3)) \end{aligned}$$

Lastly, we define the formula for the property p :

$$\text{Prop}(x, d_1, d_2, d_3) = (d_1 \wedge d_2 \wedge x) \Leftrightarrow (d_3 \wedge x)$$

Signals. Given the formulas that characterise a state machine, we define the set **Signals** that models the points in the circuit. The elements of **Signals** are functions taking state variable vectors to formulas instantiated with these variables. Specifically, for every subformula $f(s)$ of the system formulas $\text{Trans}(s, s')$ and $\text{Prop}(s)$, such that $f(s)$ is not dependent on any of the variables of s' , we add the corresponding functions f and $\neg f$ to the set. Moreover, we also add the constant signals **tt** and **ff**, for which $\text{tt}(s) = 1$ and $\text{ff}(s) = 0$.

Example 2. f_1 is a signal in Figure 1 with the definition $f_1(x, d_1, d_2, d_3) = d_1 \wedge d_2$. The negated signal $\neg f$ has the definition $\neg f_1(x, d_1, d_2, d_3) = \neg(d_1 \wedge d_2)$.

Signal Correspondence. For a given equivalence relation \equiv over the set **Signals**, we define the *signal correspondence condition*, denoted by $\text{Holds}(\equiv, s)$, as follows:

$$\text{Holds}(\equiv, s) = \bigwedge_{f \equiv g} f(s) \Leftrightarrow g(s).$$

This means that the correspondence condition for an equivalence relation is satisfied by a state when all the signals that are equivalent have the same value in

that state. We define a *signal correspondence relation* as an equivalence relation whose correspondence condition holds in all reachable states. ■

Algorithm. In order to find a signal correspondence relation, van Eijk's algorithm computes a sequence of equivalence relations \equiv_i , each being a better overapproximation of the desired relation. The sequence stops when an n is found such that \equiv_n is equal to \equiv_{n+1} .

The first approximation \equiv_0 is the equivalence relation that holds in the initial states. We can define it as follows; for all f and g , $f \equiv_0 g$ if and only if the following formula is valid:

$$\text{Init}(s_1) \Rightarrow f(s_1) \Leftrightarrow g(s_1).$$

This means that two signals are equivalent precisely if they must have the same value in all initial states. The original algorithm computes \equiv_0 by constructing a BDD for every signal, and pairwise comparing these BDDs under the assumption that the BDD for $\text{Init}(s)$ holds.

The other approximations \equiv_{n+1} for $n \geq 0$ are subsets of \equiv_n . We can define them as follows; for all f and g , $f \equiv_{n+1} g$ if and only if $f \equiv_n g$ and the following formula is valid:

$$\text{Holds}(\equiv_n, s_1) \wedge \text{Trans}(s_1, s_2) \Rightarrow f(s_2) \Leftrightarrow g(s_2).$$

This means that two signals are equivalent in the new relation, when (1) they were equivalent in the old relation and (2) they have the same value in the next state if the old relation holds in the current state. The original algorithm computes \equiv_{n+1} by pairwise comparison of the BDDs for the signals related by \equiv_n under the assumption that the BDD for \equiv_{n+1} holds.

The construction of approximations \equiv_i has the shape of an *inductive* argument; it has a base case and a step that is iterated until it is provable. The final signal correspondence relation therefore holds in all reachable states.

For a schematic overview of the algorithm, see Figure ■. At lines 5 and 12, we use the function `VALIDBDD` that checks if a formula is valid by building its BDD. At lines 6 and 13, we use `set` to modify an equivalence relation by merging the equivalence classes for f and g .

Example ■ (ctd.). The signal correspondence relation found by the algorithm for Example ■ looks as follows:

$$\{\dots, (f_1, d_3), (f_2, f_3), (p, \text{tt}), \dots\}$$

From this information it follows immediately that the property p is always true.

¹ Note that this is a slight generalisation of van Eijk's original definition ■.


```

1.   $\equiv_1, \equiv_2 := \emptyset, \emptyset$  ;
2.  -- compute first approximation
3.  for every  $f, g$  in Signals do
4.     $\text{form} := \text{Init}(s_1) \Rightarrow (f(s_1) \Leftrightarrow g(s_1))$  ;
5.    if ( $\text{VALIDBDD}(\text{form})$ ) then
6.      set  $f \equiv_2 g$  ;
7.  -- iterate until a fixed point is reached
8.  while ( $\equiv_1 \neq \equiv_2$ ) do
9.     $\equiv_1, \equiv_2 := \equiv_2, \emptyset$  ;
10.   for every  $f, g$  in Signals such that  $f \equiv_1 g$  do
11.      $\text{form} := \text{Holds}(\equiv_1, s_1) \wedge \text{Trans}(s_1, s_2) \Rightarrow (f(s_2) \Leftrightarrow g(s_2))$  ;
12.     if ( $\text{VALIDBDD}(\text{form})$ ) then
13.       set  $f \equiv_2 g$  ;
14.  return  $\equiv_1$  ;

```

Fig. 2. Van Eijk's algorithm

Remarks. The signal correspondence relation found by the algorithm sometimes implies the safety property directly. If this is not the case, then we can strengthen the transition formula $\text{Trans}(s, s')$ to a new formula $\text{Trans}(s, s') \wedge \text{Holds}(\equiv, s) \wedge \text{Holds}(\equiv, s')$. This is legal as we only are interested in transitions in the reachable state space. The new transition formula relates fewer states, and can consequently reduce the number of fixpoint iterations in conventional model checking methods.

Van Eijk's original paper presents a number of improvements of the basic method, such as *retiming* techniques that enlarge the set **Signals** so that the equivalence relation can contain more information, and *random simulation* that aims to reduce the number of pairwise comparisons by computing a better initial approximation \equiv_0 . We will not discuss these techniques here, but refer to the original paper [1].

3 Stålmarck's Method Instead of BDDs

Van Eijk's method has a number of disadvantages. First of all, sometimes it is impossible to complete the analysis as some signals in the circuit can not be represented succinctly as BDDs. Second, the algorithm always finds the largest equivalence relation, which can be unnecessarily costly for proving the property. Third, the equivalences are computed by pairwise comparisons of signals, which means we have to build a quadratic number of BDDs. We will now focus on trying to solve these problems by using a SAT method instead of BDDs.

Stålmarck's Method. Stålmarck's *saturation method* [2, 3] is a patented algorithm that is used for satisfiability checking. The method has been successfully applied in an wide range of industrial formal verification applications. The algorithm takes a set of formulas $\{p_1, \dots, p_n\}$ as input, and produces an equivalence relation over the negated and unnegated subformulas of all p_i . Two subformu-

las are equivalent according to the resulting relation only when this is a logical consequence of assuming that all formulas p_i are true. The algorithm computes the relation by carefully propagating information according to the structure of the formulas.

The saturation algorithm is parameterised by a natural number k , the *saturation level*, which controls the complexity of the propagation procedure. The worst-case time complexity of the algorithm is $O(n^{2k+1})$ in the size n of the formulas, so that for a given k , the algorithm runs in polynomial time and space. For any specific k , there are formulas for which not all possible equivalences are found, but for every formula there is a k such that the algorithm finds all equivalences. A fortunate property is that this k is surprisingly low (usually 1 or 2) for many practical applications, even for extremely large formulas.

The advantage of having control over the saturation level is that the user can make a trade-off between the running time and the amount of information that is found. A disadvantage is that it is not always clear what k to choose in order to find enough information. In contrast, finding equivalences using BDDs results in discovering either all information, or no information at all due to excessive time and space usage.

Modification of van Eijk’s Method. We now adapt van Eijk’s algorithm to use Stålmarck’s method.

To compute the initial approximation \equiv_0 , we use the saturation procedure to compute equivalence information between positive and negative subformulas of $\text{Init}(s_1)$ and $\text{Holds}(\text{Id}, s_1)$ under the assumption that both of these formulas are true. Here, Id denotes the identity equivalence relation on signals, relating f to g if and only if $f = g$. Note that $\text{Holds}(\text{Id}, s_1)$ is a valid formula, so assuming that it is true adds no real information; we just add it to the system to ensure that all subformulas that correspond to signals are present in the resulting equivalence relation. We then use the resulting information to generate the equivalence relation \equiv_0 on signals.

To improve a relation \equiv_n , we run the saturation procedure on a set of formulas that contains $\text{Holds}(\equiv_1, s_1)$, $\text{Trans}(s_1, s_2)$, and $\text{Holds}(\text{Id}, s_2)$. Again, we need the formula $\text{Holds}(\text{Id}, s_2)$ to ensure that all subformulas that correspond to signals are present. From the result we extract \equiv_{n+1} by looking at the equivalences between subformulas depending on s_2 , and taking the intersection with the original equivalence relation \equiv_n . The intersection of two equivalence relations relates two signals if both original relations relate them.

For a schematic overview of our algorithm, see Figure 4. The notation \equiv / s_1 , occurring at lines 4 and 10, turns an equivalence relation \equiv on formulas into an equivalence relation on signals, by relating two signals f and g if and only if their instantiated formulas $f(s_1)$ and $g(s_1)$ are related by \equiv .

In our modified algorithm, we have explicit control over the running time complexity of each iteration step; each step is guaranteed to take polynomial time and space. As a consequence, we do not have to worry about a possible expo-

```

1.  $\equiv_1$       :=  $\emptyset$  ;
2.  -- compute first approximation
3.  system := {Init( $s_1$ ), Holds(Id,  $s_1$ )} ;
4.   $\equiv_2$       := STÅLMARCK(system) /  $s_1$  ;
5.  -- iterate until a fixed point is reached
6.  while ( $\equiv_1 \neq \equiv_2$ ) do
7.     $\equiv_1$       :=  $\equiv_2$  ;
8.    system := {Holds( $\equiv_1$ ,  $s_1$ ), Trans( $s_1$ ,  $s_2$ ), Holds(Id,  $s_2$ )} ;
9.     $\equiv$        := STÅLMARCK(system) ;
10.    $\equiv_2$       :=  $\equiv_1 \cap (\equiv / s_2)$  ;
11. return  $\equiv_1$  ;

```

Fig. 3. Van Eijk’s algorithm using Stålmarck’s method

nential space blowup, as in the case of building BDDs. However, having this explicit control also means that we do not always compute the *largest* relation, since the saturation algorithm is possibly incomplete depending on what k we have chosen. In many cases it turns out that even for small k , the equivalence relation we compute using Stålmarck’s algorithm is still large enough to decide if the property is true or not, or to considerably reduce the number of subsequent model checking iterations.

Signal Implications. Finding equivalences between signals is a rather arbitrary choice. We could just as well try to find other information about signals that is easy to compute. For example, we can compute *implications* between signals.

An implication $f \Rightarrow g$ occurs between two signals f and g , if g must be true whenever f is true. The implications over the set of signals are interesting as they can capture *all* binary relations. The reason for this is that any formula that contains two variables can be characterised by a finite number of implications between these variables, their negations, and constants.

The presented algorithm can easily be extended to find implications between the computed equivalence classes of signals. Note that implications *within* equivalence classes do not give any information, since we know that every point in an equivalence class implies every other point in the same class. Our approach for generating the implications is simple: To begin with, we generate all the equivalence classes that hold over the reachable state space using the algorithm in Figure 4. From each equivalence class we take a representative signal. Finally, we run a modified version of the algorithm in Figure 4 that uses induction to find implications rather than equivalences between the representatives.

4 Induction

In order to further improve our adaptations of van Eijk’s method, we start by investigating another safety property verification method: *induction* [10].

Simple Induction. The idea behind simple induction is to attempt to decide whether all reachable states of the described system make the formula $\text{Prop}(s)$ true by proving that the property holds at the initial states, and proving that if it holds in a certain state, it also holds in the next state. The inductive proof is expressible in propositional logic using the following two formulas:

$$\begin{aligned}\text{Init}(s_1) &\Rightarrow \text{Prop}(s_1) \\ \text{Prop}(s_1) \wedge \text{Trans}(s_1, s_2) &\Rightarrow \text{Prop}(s_2)\end{aligned}$$

If the first formula is valid, we know that all the initial states of the system make the property true. If the second formula is valid, we know that any time a state makes the property true, all states reachable in one step from that state also make the property true. We can thus infer that all reachable states are safe.

Simple induction is not a complete proof technique for safety properties; it is easy to construct a system whose reachable states all make $\text{Prop}(s)$ true, but for which the inductive proof fails. Just take a safe system and change it by adding two unreachable states s_1 and s_2 in such a way that there is a transition between s_1 and s_2 , the formula $\text{Prop}(s_1)$ is true, and $\text{Prop}(s_2)$ is false. This system can not give a provable induction step even though all reachable states satisfy the property. A stronger proof scheme is needed for completeness.

Induction with Depth. In the step case of simple induction we prove that the property holds in the current state, assuming that it holds in the previous state. One way to strengthen the induction step is to instead assume that the property holds in the previous n consecutive states. Correspondingly, the base case becomes more demanding.

Let $\text{Trans}^*(s_1, \dots, s_n)$ be the formula that expresses that s_1, \dots, s_n are states on a path s_1, \dots, s_n , and let $\text{Prop}^*(s_1, \dots, s_n)$ be the formula that expresses that Prop is true in all of the states s_1, \dots, s_n . *Induction with depth n* amounts to proving that the following formulas are valid:

$$\begin{aligned}\text{Init}(s_1) \wedge \text{Trans}^*(s_1, \dots, s_n) &\Rightarrow \text{Prop}^*(s_1, \dots, s_n) \\ \text{Prop}^*(s_1, \dots, s_n) \wedge \text{Trans}^*(s_1, \dots, s_{n+1}) &\Rightarrow \text{Prop}(s_{n+1})\end{aligned}$$

The modified base case expresses that all states on a path with n states starting in the initial states make the property true. The step says that if $s_1 \dots s_{n+1}$ is a path where $s_1 \dots s_n$ all make Prop true, then s_{n+1} also makes Prop true. We henceforth refer to n as the *induction depth*. Note that induction with depth 1 is just simple induction.

Unique States Induction. The induction scheme with depth discovers paths to any state s in the reachable state space that makes $\text{Prop}(s)$ false: A path with n states starting from the initial states and ending in a bad state is a counterexample to base cases of depth n or larger. As we are verifying finite systems, some depth n is therefore sufficient to discover all bugs.

Unfortunately the scheme is still not complete; it is possible to construct a safe system where the induction step fails for any depth n . Just take any safe system

and change it by adding two unreachable states s_1 and s_2 , so that the property holds in s_1 , s_1 can both reach itself and s_2 , and the property fails in s_2 . Then there exist a counterexample for every depth n that loops $n - 1$ times in s_1 , and then visits s_2 .

However, a state that is reachable from the initial states must be reachable by at least one path that only contains *unique* states. Therefore, we can add a formula $\text{Uniq}(s_1, \dots, s_n)$ to the induction step that expresses that s_1, \dots, s_n are different from each other. This restriction on the shape of considered paths makes it impossible to generate counterexamples of arbitrary length from loops in the unreachable state space. The induction step now becomes:

$$\text{Prop}^*(s_1, \dots, s_n) \wedge \text{Trans}^*(s_1, \dots, s_{n+1}) \wedge \text{Uniq}(s_1, \dots, s_{n+1}) \Rightarrow \text{Prop}(s_{n+1})$$

The result is a complete scheme for verifying safety properties of finite systems: If there are paths in the unreachable state space that falsely make the induction step unprovable, they are ruled out from consideration by some induction depth n . However, a major problem is that this n can be extremely large for some verification problems, and that it is often difficult to predict what n is needed.

5 Stronger Induction in van Eijk's Method

We will now make use of the insight into induction methods we gained in the previous section. The underlying proof method that van Eijk's algorithm uses to find equivalences that always hold in the reachable state space is simple induction. Recall that we have demonstrated that this proof technique is too weak to prove all properties that hold globally in the reachable states. Consequently there are circuits that contain useful equivalences that van Eijk's original algorithm misses due to the incompleteness of its underlying proof method.

Generalisation to Completeness. We can make van Eijk's original algorithm complete by modifying our implementation to use unique states induction with depth n rather than simple induction. In the base case of the algorithm, we compute an equivalence relation on signals that hold in the first n states on paths from the initial states. In the algorithm step, we assume that our most recently computed signal equivalence relation holds in the first n of $n + 1$ consecutive unique states, and derive the subset of the signal equivalences that necessarily holds in state $n + 1$.

For a detailed description of the resulting algorithm, see Figure 1. We use the notation $\text{Holds}^*(\equiv, s_1, \dots, s_n)$, occurring at lines 3 and 9, as a shorthand for $\text{Holds}(\equiv, s_1) \wedge \dots \wedge \text{Holds}(\equiv, s_n)$. The algorithm for finding implications between signals is modified in an analogous way.

We can now discover all equivalences that hold globally in the state space. In particular, if a safety property holds in all reachable states, there exists a saturation level and an induction depth that is sufficient to discover that the corresponding signal is equivalent to the true signal.

```

1.   $\equiv_1$       :=  $\emptyset$  ;
2.  -- compute first approximation
3.  system := {Init( $s_1$ ), Trans*( $s_1, \dots, s_n$ ), Holds*(Id,  $s_1, \dots, s_n$ )} ;
4.   $\equiv$       := STÅLMARCK(system) ;
5.   $\equiv_2$       := ( $\equiv / s_1$ )  $\cap \dots \cap (\equiv / s_n)$  ;
6.  -- iterate until a fixed point is reached
7.  while ( $\equiv_1 \neq \equiv_2$ ) do
8.     $\equiv_1$       :=  $\equiv_2$  ;
9.    system := {Holds*( $\equiv_1, s_1, \dots, s_n$ ), Trans*( $s_1, \dots, s_{n+1}$ ),
               Holds(Id,  $s_{n+1}$ ), Uniq( $s_1, \dots, s_{n+1}$ )} ;
10.    $\equiv$       := STÅLMARCK(system) ;
11.   ( $\equiv_2$ )    :=  $\equiv_1 \cap (\equiv / s_{n+1})$  ;
12.  return  $\equiv_1$  ;

```

Fig. 4. The adaption of the algorithm for depth n unique states induction

As an additional benefit, the possibility to adjust both the saturation level and the induction depth allows a high degree of control over how much work is spent on discovering equivalences. We can now increase the number of equivalences that can be discovered for a fixed saturation level by increasing the induction depth; this can be useful as an increase in saturation level means a big change in the time complexity of the algorithm.

We note that the idea of using stronger induction not is restricted to our SAT-based version of van Eijk’s algorithm; the original BDD-based algorithm can also be made complete by stronger induction.

6 Approximations

In this section we show that van Eijk’s algorithm is an *approximative forwards reachability* analysis. We then use this insight to derive an analogous backwards approximative analysis, and combine the two algorithms into a mutual improvement algorithm.

The Forwards Reachability View. Figure 1 shows the shape of a standard forwards reachability analysis, where we use INIT to denote the set of initial states, and the operation POSTIMAGE to compute postimages (the postimage of a set of states S is the set of states that can be reached from S in one transition). We now demonstrate that van Eijk’s algorithm performs such a forwards analysis approximatively, in the sense that it is a variant of the standard analysis where INIT has been replaced with an overapproximation, and the exact operations \cup and POSTIMAGE have been replaced by overapproximative operators.

Van Eijk’s algorithm computes a sequence of relations \equiv_i . Each of the corresponding formulas Holds(\equiv_i, s) can be seen as the characterisation of a set of states A_i .

```

1.  $n, S_0 := 0, \text{INIT}$  ;
2. loop
3.    $S_{n+1} := \text{POSTIMAGE}(S_n) \cup S_n$  ;
4.    $n := n + 1$  ;
5. until ( $S_{n+1} \neq S_n$ ) ;
6. return  $S_{n+1}$  ;
    
```

Fig. 5. A standard forwards reachability algorithm

In the base case, the algorithm computes the binary relation \equiv_0 that holds between points in all the initial states. The formula $\text{Holds}(\equiv_0, s)$ will therefore be valid for every state s that makes $\text{Init}(s)$ valid, and possibly for some other states. A_0 is consequently a superset of the initial states.

In the step, the algorithm computes the subrelation \equiv_{n+1} of \equiv_n that provably holds after a transition under the assumption that \equiv_n holds before the transition. Every state s that is reachable in one transition from a state in A_n therefore makes $\text{Holds}(\equiv_{n+1}, s)$ valid. But \equiv_{n+1} is a subrelation of \equiv_n , so every state s in A_n also satisfies $\text{Holds}(\equiv_{n+1}, s)$. Consequently, the step operation corresponds to computing A_{n+1} as the overapproximative union of A_n and the overapproximative postimage of A_n .

Finally, the algorithm checks whether \equiv_{n+1} is the same relation as \equiv_n . This corresponds to checking whether $A_{n+1} = A_n$. If this is the case, the algorithm terminates, otherwise the step is iterated.

Approximative Backward Analysis. It is well known that forwards reachability analysis has a dual analysis called *backwards* reachability analysis [1]. We can perform the backward analysis using the forwards algorithm by modifying the characterisation of the underlying system in the following way:

$$\begin{aligned}
 \text{Init}'(s) &= \neg \text{Prop}(s) \\
 \text{Trans}'(s, s') &= \text{Trans}(s', s) \\
 \text{Prop}'(s) &= \neg \text{Init}(s)
 \end{aligned}$$

The result of the computation is the set of states that are backwards reachable from the bad states—the states where the property does not hold.

We can use the system transformation together with any of our variants of van Eijk’s algorithm. In particular, we can compute a relation \equiv that characterises an overapproximation of the states that are backwards reachable from the states that make $\text{Prop}(s)$ false. Analogously to the forwards algorithm, the system is safe if $\text{Holds}(\equiv, s)$ implies the safety property, which in this case corresponds to that no initial state is in the overapproximation of the set that can be backwards reached from the bad states. Also, if we do intersect the initial states, we can still use the approximation to constrain the transition relation in order to reduce the number of necessary iterations of an exact forwards reachability algorithm.

Mutual Improvement. The new approximate backward algorithms can be very useful on their own. However, there exists a general way of enhancing approximative reachability analyses that improves matters further [5].

The idea is to first generate the overapproximation of the reachable states. If the corresponding set has an empty intersection with the bad states, we are done. If it has a nonempty intersection, there are two possible reasons: Either the system is unsafe, or the approximation is too coarse. Regardless of which is the case, we know that the only possible bad states we can reach are those that are contained in the intersection. We can therefore take the intersection to be our new bad states.

But now we can apply approximate backwards reachability analysis from the new bad states. If we do not intersect the initial states, the system must be safe. If we do, we can take the intersection to be the new initial states and restart the whole process. The algorithm terminates if we generate the same overapproximations twice, as this implies that no further improvement is possible.

The resulting mutually improved overapproximations are always at least as good as the original overapproximations, and they can sometimes be substantially better as we demonstrate in the next section.

7 Experimental Results

In this section, we present a number of experiments we have done using a prototype implementation of our variants of van Eijk's algorithm. We compare our results against the results of three other methods. The first two are reachability analysis and unique states induction, as implemented in the SAT-based model checking workbench FixIt [4]. The third method is BDD-based model checking, as implemented in the verification tool VIS version 1.3. In the experiments with VIS we have used dynamic variable reordering and experimented with different partitionings [1]. All running times are measured on a 296 MHz Ultrasparc-II with 512 MB memory. The results are displayed in Table 1.

The motivation for the choice of benchmarks is as follows. We have chosen one industrial example, one example that is difficult to represent with BDDs, and one example that belongs to the easy category for BDDs.

The Lalita Example. The Lalita example is an industrial telecommunications example from Lucent Technologies that was one of the motivations for the research presented in this paper. We received the example as a challenge from Prover Technology, a Swedish formal verification company. It was given to us as a black-box problem; we had no information about the structure of the system. The design was already known to be within reach of BDD technology, but not all of the properties were possible to verify using unique states induction.

² We have also tried approximate model checking in VIS, but there appears to be a bug in the implementation which makes it unsound.

Property	FixIt Reach.	FixIt Induct.	VIS	Our Method
Lalita, nr. 2	0.3	0.2	219.9	2.3 ^a
7	41.8	0.2	207.3	2.2 ^a
10	[>15min]	[>15min]	86.6	9.7 ^b
11	[>15min]	[>15min]	199.3	2.1 ^a
Butterfly, size 2	0.1	1.0	0.3	0.1 ^a
4	16.6	[>15min]	2.0	0.1 ^a
16	[>15min]	—	[>15min]	1.4 ^a
64	—	—	—	37.4 ^a
Arbiter	2.5	[>15min]	2.1	76.9 ^c

^a with equivalences, ^b with mutual improvement, ^c with implications

Table 1. Experimental results (times are in seconds).

When we attempted to verify the design using SAT-based reachability analysis, the representations became excessively large due to the computation of pre- and postimages.

The design contains 178 latches. The problem comes with thirteen safety properties that should be verified; we present the four most interesting properties: the two properties that were most difficult for VIS (2 and 7), one of the two properties that are hard for induction (11), and the property that was hardest for our methods (10).

All of the properties except property 10 and 11 can be done using SAT-based induction. However, we can verify or refute every property except property 10 directly using our forwards equivalence algorithm. Property 10 is verified using one iteration of mutual improvement of the computed equivalences. As the table demonstrates our analyses are a factor 10 to 100 faster than BDD-based verification in VIS.

The Butterfly Circuits. This family of benchmarks arose when we were designing sorting circuits for implementation on an FPGA. The problem is to decide whether a butterfly network containing reconfigurable sequential components is equivalent to an optimised version where the components have been shifted around. When we attempted to verify the circuits we discovered that standard algorithms could not handle circuits of any reasonable size. In particular, BDD-based methods did not work because the BDDs representing the circuits became too large.

The model checking algorithms in VIS are unable to verify larger networks than size 4 in a reasonable amount of time and space. SAT-based reachability analysis and induction are also unable to cope with larger instances of the circuits.

However, the forwards equivalence algorithm handles all the sizes we have tried in less than 40 seconds.

The Arbiter. This example is a simple benchmark from the VIS distribution. The arbiter controls three clients that compete for bus access. We verify the property of mutual exclusion.

The problem is easy both for BDDs and SAT-based symbolic reachability analysis, but can not be done using unique states induction. The example clearly demonstrates that finding implications between equivalence classes can be stronger than only computing equivalences: Our equivalence based analysis alone is unable to verify the design in a reasonable amount of time, but we can verify the design in 77 seconds by computing implications between the equivalence classes.

8 Related Work

The first approach in the literature to apply SAT-based techniques to model checking was Bounded Model Checking [10]. Bounded model checking of safety properties corresponds to searching for bugs by attempting to prove the base case only of induction with depth. Certain bugs that are hard to find using BDD-based model checking can be found very quickly in this way. In order to effectively also prove safety of systems, standard symbolic reachability analysis was adapted to use SAT-solvers [11] which resulted in the analysis implemented in FixIt. Currently, interesting work is being done on combinations of SAT-solvers and BDDs for model checking [12].

The idea to use approximate analyses to generate semantic information from systems originally comes from the field of program analysis. Many different such analyses can be seen as abstract interpretations [13]. In particular, Halbwachs et al. [14,15] have used abstract interpretation techniques to generate linear constraints between arithmetic variables that always holds in the reachable state space of synchronous programs and timed automata. This information is used both for compilation purposes and for verification. The same techniques are used for generating strengthenings in the STeP system [16] that is targeted towards deductive verification of reactive programs.

The main differences between our work and the work on synchronous programs and STeP, is (1) that the analyses we present here are specially designed for generating information about *gate level circuits* rather than programs, (2) that we focus specifically on simple relations between boolean signals, and (3) that we use Stålmarck's saturation method as a possibly incomplete but fast method for generating the relations. Also, we generate information while keeping in mind that we can apply an exact analysis later, and we have consequently optimised the algorithms for quickly generating information. In the case of synchronous program verification and STeP, a precise analysis is not even possible in general as infinite state systems are addressed.

Dill and Govindaraju [7] have developed a method for performing BDD-based approximate symbolic model checking based on overlapping projections. Their idea is to alleviate BDD blow-up by representing an overapproximation of a set of states S as a vector of BDDs, where each individual BDD characterises the relation in S between some subset of the state variables. The conjunction of the BDDs represents an overapproximation of the underlying set. The main difference between our approach and theirs, is that we consider some particular relations between *all* pairs of signals, while they consider all relations between a number of subsets of state variables. Also, the user of Dill and Govindaraju's method must manually choose the subsets of state variables to build BDDs for, whereas our methods are fully automatic.

9 Conclusions and Future Work

We have taken an existing BDD-based verification method which finds equivalent points in a circuit, and adapted it to use Stålmarck's method instead of BDDs. Then, we strengthened the resulting algorithm by combining it with recently developed induction techniques. We also discussed how the algorithm can be improved by computing implications rather than simple equivalences between points. Lastly, we observed that the algorithm can be transformed into a mutual improvement approximative reachability analysis.

The resulting collection of new algorithms can be seen as SAT-based improvements of van Eijk's original algorithm, where we use stronger inductive methods. Viewed from this angle, we have made van Eijk's method complete and provided a more fine-grained tuning between the time used and the information found.

Viewing our work in a different way, we can say that we have improved an inductive method by using van Eijk's algorithm to find equivalences. In some cases, such as the butterfly examples (see Section 4), unique states induction needs an exponentially larger induction depth than our improved analyses.

We believe the proposed methods work well for several reasons. First of all, van Eijk's original idea of finding equivalences of points in the circuit makes it very hard for properties to "hide" deep down in the logic of a circuit. Comparing this with problems occurring with methods that only look at state variables (such as conventional model checking methods) or methods that only look at the outputs (such as inductive methods) clearly shows that this is a desirable thing to do.

Second, the use of Stålmarck's saturation algorithm forms a natural fit with van Eijk's original algorithm. The possibility of controlling the saturation level pays off especially in systems where it is hard to find *all* equivalences, but sufficient to find some. Stålmarck's algorithm is also rather robust in the number of variables used in formulas.

Third, inductive methods perform well because they do not need any iteration or complicated quantification. Unfortunately, when we prove partial properties of

systems, or when we prove properties about systems with a lot of logic between the latches and the property, induction performs poorly because the induction hypothesis is not strong enough to establish the inductive step. In this case, finding equivalence or implication information is just the right thing to do, because it strengthens the induction hypothesis, and provides direct information not only about the latches, but about all points in the circuit.

For future work, we would like to investigate other signal relations than equivalences and implications. General relations over three variables is a candidate, although it is not clear how to represent the found information. Furthermore, we are interested in extending the proposed algorithms to work with other properties than just safety properties. Lastly, we would like to extend the presented ideas to the verification of safety properties of synchronous reactive systems; for example, systems implemented in the programming language Lustre [14]. In order to do this, we need to add support for integer arithmetic and to investigate how Halbwachs's ideas [15] can be combined with our analyses.

Acknowledgements

Many thanks to Niklas Sörensson who implemented a large part of the algorithms and benchmarks, as well as to Niklas Eén, who helped with running some of the benchmarks. We also thank Byron Cook, Koen van Eijk, Carl-Johan Lillieroth, Gordon Pace, Mary Sheeran, and Satnam Singh for their useful comments on earlier drafts of this paper.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. TACAS '00, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '99, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.
3. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
6. C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Conf. on Design, Automation and Test in Europe*, 1998.
7. S. G. Govindaraju and D. L. Dill. Approximate symbolic model checking using overlapping projections. In *Electronic Notes in Theoretical Computer Science*, July 1999. Trento, Italy.

8. N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur, Belgium, September 1994. LNCS 864, Springer Verlag.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
10. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.
11. Z. Manna and the STeP group. STeP: The Stanford Temporal Prover. Technical report, Computer Science Department, Stanford University, July 1994.
12. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.
13. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000.
14. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. *Swedish Patent No. 467076 (1992)*, *U.S. Patent No. 5 276 897 (1994)*, *European Patent No. 0403 454 (1995)*, 1989.
15. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. 12th Int. Conf. on Computer Aided Verification*, 2000.

Scalable Distributed On-the-Fly Symbolic Model Checking

Shoham Ben-David², Tamir Heyman^{1,2}, Orna Grumberg¹, and Assaf Schuster¹

¹ Computer Science Department, Technion, Haifa, Israel

² IBM Haifa Research Laboratories, Haifa, Israel

Abstract. This paper presents a scalable method for parallel symbolic on-the-fly model checking on a distributed-memory environment of workstations. Our method combines a parallel version of an on-the-fly model checker for safety properties with a scalable scheme for reachability analysis. The extra load of storage required for counter example generation is evenly distributed among the processes by our memory balancing. For the sake of scalability, at no point during computation the memory of a single process contains all the data from any of the cycles. The counter example generation is thus performed through collaboration of the parallel processes. We develop a method for the counter example generation keeping a low peak memory requirement during the backward step and the computation of the inverse transition relation.

We implemented our method on a standard, loosely-connected environment of workstations, using a high-performance SMV-based model checker. Our initial performance evaluation using several large circuits shows that our method can check models that are too large to fit in the memory of a single node. Our on-the-fly approach may find counter examples even when the model is too large to fit in the memory of the parallel system.

1 Introduction

A model checking algorithm gets a system model and a specification written as a temporal logic formula. It returns ‘true’ if the model satisfies the formula and returns ‘false’ otherwise. In the latter case it also provides a counter example to demonstrate why the model does not satisfy the formula. The counter example feature is extremely important in the debugging process of the system.

Model checking tools have been successful in finding subtle errors in complex designs. Their main drawback, however, is their high space requirements that limits their applicability to large designs. Many approaches to reducing the space requirements have been investigated. One of the most successful approaches is *symbolic model checking* [6] in which the model as well as intermediate results are represented using Binary Decision Diagrams (BDDs) [4].

A different approach is *on-the-fly model checking* that develops parts of the model as needed for checking the formula. Usually, the check is guided by an automaton that *monitors* the behavior of the system in order to detect erroneous behaviors. On-the-fly algorithms [10][7][3] are usually based on a depth first search (DFS) traversal of the state space and therefore do not combine well with BDD-based methods. The method in [2],

on the other hand, reduces model checking to reachability analysis and is therefore successfully implemented with BDDs. Furthermore, their monitoring automaton is linear in the size of the formula whereas other methods use exponential size automata. The method can handle a large class of safety properties, including RCTL (see appendix A for a detailed description).

In [7,16,15,18,11], other approaches to reducing the space requirements were introduced. These studies suggest to partition the work into several tasks: [18] shows how to parallelize an explicit-state model checker that does not use symbolic methods; [7,16,15] use a single computer that handles one task at a time, while the other tasks are kept in an external memory. [11] suggests a distributed, symbolic algorithm for reachability analysis that works on a network of processes with distributed memory. The algorithm in [11] achieved an average memory scale-up of 55 on 130 processes. Thus, it made it possible to handle designs that could not fit in a single machine.

In this work we combine the approaches of [2] and [11], obtaining a distributed, symbolic, on-the-fly model checking that can handle very large designs. We also provide a counter example that is computed on the distributed state space by several processes, without ever holding the whole set of states in one process.

Producing the counter example requires additional storage of sets of states during reachability, one set for each cycle. In order to balance this extra storage across the processes we apply a slicing function which may vary from one cycle to another. This turns the efficient distributed production of a counter example to be somewhat tricky: we need to track the cycles backwards while switching different slices and keeping the peak temporal space at a low level.

We implemented our method within the high-performance verification tool RuleBase [1] of IBM, Haifa. Our initial performance results, measured on a distributed, non-dedicated system of 32 standard workstations and using a slow network interconnection, show that our method scales well. Large examples that could not fit into the memory of a single machine terminate using the parallel system. The parallel system appears to be balanced with respect to memory utilization, and the network resource does not become a bottleneck.

In addition to the above, we show that our method is more effective when applied to on-the-fly model checking including counterexample generation than when applied to reachability analysis. There are two main reasons for this phenomenon. First, note that counter example generation generally requires saving sets of states which consume more space. Effective splitting and balancing the extra space across the distributed system enhances scalability. Second, the parallel system, even when failing to complete reachability to the fix-point, is commonly able to proceed reachability for several steps beyond the point reached by a single machine. This improves the chances for our on-the-fly model checking to find an error state during those steps.

The rest of the paper is organized as follows. Section 2 describes the sequential on-the-fly algorithm for checking RCTL safety properties. Section 3 presents our distributed on-the-fly model checking scheme. Section 4 provides our initial performance evaluation. Finally, we give our conclusions in Section 5.

2 The Sequential On-the-Fly Algorithm

In this section we describe the main characteristics of the sequential on-the-fly model checking algorithm presented in [2]. This algorithm is the basis for our distributed algorithm.

Given a system model M and an RCTL formula φ , an automaton (satellite) \mathcal{A} is constructed and combined with M . \mathcal{A} monitors the behavior of M . If it detects an erroneous behavior it enters a special error state and stays there forever. Thus, M satisfies φ iff the combination of M and \mathcal{A} , $M \times \mathcal{A}$, does not reach an error state. In order to check that M satisfies φ we therefore run a reachability analysis on $M \times \mathcal{A}$ that constantly checks whether an error state has been encountered. The algorithm traverses the (combined) model using breadth first search (BFS). Starting from the set of initial states, at each iteration it constructs a *doughnut*, which is the set of newly found states that have not been found in previous iterations. The doughnuts are kept for later use in the generation of the counter example. Having to keep the doughnuts, the space requirements of this algorithm exceed those of (pure) reachability analysis.

The model checking phase terminates successfully if all reachable states have been traversed and no error state has been found. If at any stage an error state is found, the model checking phase stops and the generation of counter example starts.

The counter example is a sequence of states leading from an initial state to an error state. It is generated starting from an error state and going backwards, following the doughnuts produced and stored by the model checking algorithm.

Figure 1 presents the sequential algorithm for on-the-fly model checking, including counter example generation. This algorithm for constructing the counterexample is based on the one in [8]. Lines 1–9 describe the model checking phase. At each iteration i , the set of new states that have not been reached before is kept in doughnut S_i .

The algorithm terminates either if no new states are found ($\text{new} = \emptyset$), in which case it announces success, or if an error state is found ($\text{new} \cap \text{error} \neq \emptyset$), in which case it announces failure.

In lines 16–22 the counter example $\text{Ce}_0, \dots, \text{Ce}_k$ is generated. The counter example is of length $k + 1$ (line 14) since an error state was first found in the k -th iteration. We choose $\text{Ce}_k \in S_k$ to be one of the reached error states. Having already chosen a state $\text{Ce}_i \in S_i$, we compute the set of bad states by finding the set of predecessors for Ce_i , $\text{pred}(\text{Ce}_i)$, and intersecting it with the doughnut S_{i-1} (line 19). Since each state in S_i is a successor of some state in S_{i-1} , the set *bad* will not be empty. We now choose Ce_{i-1} from *bad*.

The generation of the counter example is completed when Ce_0 is chosen.

3 The Distributed Algorithm

The distributed algorithm for on-the-fly model checking also consists of two phases: the model checking phase and the phase in which the counter example is generated.


```

1 reachable = new = initialStates;
2 i = 0;
3 while ((new  $\neq \emptyset$ ) && (new  $\cap$  error =  $\emptyset$ )) {
4    $S_i$  = new;
5   i = i+1;
6   next = nextStateImage(new);
7   new = next  $\setminus$  reachable;
8   reachable = reachable  $\cup$  next;
9 }
10 if (new =  $\emptyset$ ) {
11   print ``formula is true in the model``;
12   return;
13 }
14 k = i;
15 print ``formula is false in the model, failed at cycle k``;
16 bad = new  $\cap$  error;
17 while (i>=0) {
18    $Ce_i$  = choose one state from bad;
19   if (i>0) bad=pred(state) $\cap S_{i-1}$ ;
20   i = i-1;
21 }
22 print ``counter examples is:``  $Ce_0 \dots Ce_k$ ;

```

Fig. 1. Sequential algorithm for on-the-fly model checking, including counter examples generation

3.1 Distributed Model Checking

The distributed algorithm is composed of an initial sequential stage, and a parallel stage. In the sequential stage, the reachable states are computed on a single process as long as memory requirements are below a certain threshold. When the threshold is reached, the state space is partitioned into k slices (the algorithm for slicing is described in [11]) and k processes are initialized. Each process is informed of the slice it *owns*, and of the slices owned by each of the other processes (which are *non-owned* by this process). The process receives its own slice and proceeds to compute the reachable states for that slice in iterative BFS steps.

During a single step each process computes the set *next* of states that are directly reached from the states in its *new* set. The *next* set contains owned as well as non-owned states. Each process splits its *next* set according to the k slices and sends the non-owned states to their corresponding owners. At the same time, it receives set of states it owns from other processes.

The model checking phase for one process P_j is given in lines 1-13 of Figure 3.2. Lines 1-3 describe the setup stage: the process receives the slice it owns, and the initial sets of states it needs to compute from. Lines 5-17 describe the iterative computation.

A *distributed termination detection* (line 5) is used in order to determine when this phase should end. *All* processes should end this phase if one of two conditions hold:

Either none of the processes found a new state or one of them found an error state. In the first case the specification has been proven correct and the algorithm terminates. If the second case the specification is false and all processes proceed to the next phase in which a counter example will be generated.

The distributed model checking is different from the sequential one in the following points:

- The set `next` is modified (lines 9-10) as the result of communication with the other processes, and stricted to include only owned states.
- Distributed termination detection is used.
- Each process P_j stores, for each doughnut i , the slice of the doughnut $S_{(i,j)}$ it owns.

One of the most important factors in making our distributed algorithm effective is *memory balancing* which keeps approximately equal memory requirement across the processes during the entire computation. Balance is maintained during the whole computation by pairing large slices with small ones and re-slicing their union in a balanced way.

It should be noted that as a result of memory balancing, a process owns (and stores) a different slice of the doughnuts in different iterations. This, however, does not effect the correctness of our distributed generation of the counter example. To guarantee that the distributed algorithm always finds a (correct) counter example, all we need is the following property, which is true by the construction:

$$S_i = \bigcup_j S_{(i,j)}, \quad (1)$$

where S_i is the doughnut computed by the sequential algorithm at iteration i .

3.2 Distributed Counter Example Generation

In this section we present an algorithm which generates a counter example. The algorithm uses the doughnut slices that are stored in the memory of the processes.

The algorithm consists of *local phases* and *coordination phases*. In the local phase all processes run in parallel. Each process executes the sequential algorithm for counter example generation until it cannot proceed any more. A process may get stuck after producing a suffix $Ce_i \dots Ce_k$ of the counter example if it cannot find a predecessor for Ce_i in its own slice of the $(i-1)$ -th doughnut.

However, by property (II) and by the fact that each element in S_i has a predecessor in S_{i-1} , we know that there must be a process that has a predecessor for Ce_i in its slice of the $(i-1)$ -th doughnut.

We would like to re-initiate the local phase in all processes from the largest suffix produced so far. In the coordination phase, a process which produced a largest suffix is selected. If this suffix is complete, i.e., it contains all of $Ce_0 \dots Ce_k$, then the process prints its counter example and all processes terminate. Otherwise, the process broadcasts its suffix together with its iteration number to all other processes. Each process updates its data accordingly and re-initiates the local phase from that point.

Lines 18-35 of Figure 3.2 describe the algorithm. Lines 22-26 contain the local phase while lines 27-35 contain the coordination phase. The algorithm uses the following variables. $myId$ is the index of the process ($myId=j$ for process P_j). The coordination phase, starts by choosing the smallest iteration number $minIte$ and then, among the processes with that number, the smallest index $minProc$ of such a process.

```

1 mySlice = receive(fromSingle);
2 reachable = receive(fromSingle);
3 new = receive(fromSingle);
4 i = 0;
5 while (Termination(new,error)==0) {
6      $S_{(i,j)}$  = new;
7     i = i+1;
8     next = nextStateImage(new);
9     next = sendRecieveAll(next);
10    next = next  $\cap$  mySlice;
11    new = next  $\setminus$  reachable;
12    reachable = reachable  $\cup$  next;
13 }
14 if (new =  $\emptyset$ ) {
15     print ``formula is true in the model``;
16     return;
17 }
18 k = i;
19 print ``formula is false in the model, failed at cycle k``;
20 bad = new  $\cap$  error;
21 while (i>=0) {
22     while ((i>=0) &&(bad  $\neq \emptyset$ )) {
23          $Ce_i$  = choose one state from bad;
24         if (i>0) bad=pred( $Ce_i$ ) $\cap S_{(i-1,j)}$ ;
25         i = i-1;
26     }
27     (minIte,minProc)=chooseMinIteFromAll(i,myId);
28     i = minIte;
29     if (i<0) {
30         if (myId == minProc)
31             print ``counter examples is:``  $Ce_0 \dots Ce_k$ ;
32         return;
33     }
34      $Ce_{i+1} \dots Ce_k$ =broadcast(minProc,  $Ce_{i+1} \dots Ce_k$ );
35     bad=pred( $Ce_{i+1}$ ) $\cap S_{(i,j)}$ ;
36 }

```

Process P_j in the distributed algorithm for on-the-fly model checking, including the generation of a counter example.

3.3 Reducing Peak in Memory Requirement

The generation of the counter example involves the computation of the sets $\text{bad} = \text{pred}(\text{Ce}) \cap S_{(i,j)}$, in which the doughnut slice $S_{(i,j)}$ is intersected with the set of predecessors of the state Ce (lines 24, 35). The BDDs for Ce and bad are usually small. However, a very large peak in memory requirement may occur by intermediate BDDs obtained during the computation of bad . In particular, the BDD for pred might be extremely big. This phenomenon can be viewed for instance in example GXI (Figure 7), where a significant increase in the memory use occurs once the computation of the counter example starts.

Examining more closely the computation of bad we notice that by changing the order of operations we can obtain smaller intermediate BDDs, thus reduce the memory peak. This can be done by first restricting the transition relation of our model to the doughnut slice $S_{(i,j)}$ and only then using it to compute $\text{pred}(\text{Ce})$. Since our implementation is based on partitioned transition relation [5], we actually restrict each one of the partitions to the doughnut slice.

To make this precise, we define the operations we perform by means of boolean functions (represented as BDDs). Assume that our model consists of a set of boolean variables V . The boolean function $TR(V, V')$ represents the transition relation of the model, where V and V' represent the current and next state, respectively.

Let $\text{Ce}(V)$ be the boolean function for the singleton set consisting of the state Ce , and $S_{(i,j)}(V)$ be the boolean function for the slice $S_{(i,j)}$. Then the computation of bad can be described by:

$$\exists V' [TR(V, V') \wedge \text{Ce}(V')] \wedge S_{(i,j)}(V) \quad (2)$$

Our transition relation is partitioned, which means that it consists of partitions $N_n(V, V')$ so that $TR(V, V') = \bigwedge_n N_n(V, V')$. Consequently, the previous expression can be rewritten as:

$$\exists V' [\bigwedge_n N_n(V, V') \wedge \text{Ce}(V')] \wedge S_{(i,j)}(V) \quad (3)$$

Since $S_{(i,j)}(V)$ does not depend on V' it can be moved into the scope of the quantifier, resulting in an equivalent expression:

$$\exists V' [\bigwedge_n (N_n(V, V') \wedge S_{(i,j)}(V)) \wedge \text{Ce}(V')] \quad (4)$$

This expression describes the computation in which, first, each partition of the transition relation is restricted to the doughnut slice $S_{(i,j)}$, and then the predecessors of Ce are computed.

This computation can be made more efficient by using the *simplify-assuming* technique [9]. Instead of intersecting each $N_n(V, V')$ with $S_{(i,j)}(V)$ we simplify $N_n(V, V')$ assuming $S_{(i,j)}(V)$ and intersect only the final result with $S_{(i,j)}(V)$.

The improvement described above uses precise information in order to restrict the partitions of the transition relation. This requires, however, to compute a different restriction (with respect to different slice) in each step of the counter example generation.

We next suggest a different restriction that can be computed only once for each process. Process P_j restricts $N_n(V, V')$ to U_j , where $U_j = \cup_i S_{(i,j)}$ is the union of all the doughnut slices owned by P_j . This restriction is expected to give a similar improvement in space reduction.

We now prove that the computation of bad with the new restriction results in exactly the same set of states. Since $S_{(i,j)} \subseteq U_j$, the expression in (4) is equivalent to:

$$\exists V' \left[\bigwedge_n (N_n(V, V') \wedge U_j(V) \wedge S_{(i,j)}(V)) \wedge Ce(V') \right] \quad (5)$$

This, in turn, is equivalent to:

$$\exists V' \left[\bigwedge_n (N_n(V, V') \wedge U_j(V)) \wedge Ce(V') \right] \wedge S_{(i,j)}(V) \quad (6)$$

This final expression describes the computation in which each partition is restricted to U_j and then used in finding the predecessors of Ce .

Note that, $N_n(V, V') \wedge U_j(V)$ can be computed only once at the beginning of the counter example generation. Here again a better improvement may be obtained by simplifying $N_n(V, V')$ assuming $U_j(V)$.

We next suggest an orthogonal improvement that exploits the fact that we compute the set of predecessors of a singleton ($Ce(V')$ contains only one state Ce). We replace the intersection of $N_n(V, V')$ and $Ce(V')$ by assigning Ce to V' in N_n . The existential quantifier is then redundant and can be removed. Applying these operation, equation (3) can first be rewritten as

$$\exists V' \left[\bigwedge_n (N_n(V, Ce)) \right] \wedge S_{(i,j)}(V) \quad (7)$$

Removing the redundant quantification, we get

$$\bigwedge_n (N_n(V, Ce)) \wedge S_{(i,j)}(V) \quad (8)$$

To combine the two optimizations we can compute (3) as

$$\bigwedge_n (N_n(V, Ce) \wedge S_{(i,j)}(V)) \quad (9)$$

This expression describes the computation in which, first the state Ce is assigned to each partition of the transition relation, the result is then restricted to the doughnut slice $S_{(i,j)}$, and finally the intersection of all the partitions is computed.

4 Experimental Results

In this section we report initial performance evaluation of our approach. We implemented our On-The-Fly model checker and embedded it in an enhanced version of McMillan's SMV [14], developed at IBM Haifa Research Laboratory [1].

Our parallel testbed includes 32 RS6000 machines, each consisting of a 225MHz PowerPC processor and 512MB memory. The communication between the nodes consists of a 16Mbit/second token ring. The nodes are non-dedicated; i.e., they are mostly workstations of employees who would often use them (and the network) at the same time that we ran our experiments.

We experimented using two of the largest circuits we found in the benchmarks IS-CAS89 +addendum'93. In order to test the counter examples generation we used common properties that are often tested when verifying hardware designs. We also used two large examples (BIQ and GXI) which are components of IBM's Gigahertz processor. For these examples we used the original properties. We mapped properties to automata using the IBM implementation as described in [2]. Characteristics of the circuits and the automata are given in Figure 2.

4.1 Parallel On-the-Fly Model Checking – Space Reduction

We now present the results for On-The-Fly model checking of the benchmark suit using our 32 machine testbed. Figures 3 to 8 summarize the memory usage, giving the store size and peak usage for every step. Each of the graphs compares the memory usage in the single-machine execution to that of the parallel system. For the parallel system we give both average and highest (peak) memory utilization in any of the machines.

We give examples for four models and six properties. For two of the models, BIQ and S1423, two properties are checked: one that overflows using a single machine, and another one which completes the computation even when using only a single machine.

As can be seen in Figure 3, an overflow occurs at cycle 15 while searching for an error state in BIQ using a single machine. The overflow occurs because counter example generation (CE) requires saving the doughnuts which consume a lot of memory. In contrast, the parallel algorithm does not overflow, and finds the error state in BIQ at cycle 17, at which point counter example generation begins.

At the first counter example cycle we see a drop in the memory store size. This drop is characteristic to all examples, and is caused by two factors. First, the transition relation computations during a backward step are usually simpler than those performed during a forward step and require less memory. The reason for that is the relative simplicity of the relation consisting of a single origin (the last state in the CE found so far). Second, the set of reachable states can be released since it is not needed for the counter example generation.

The parallel algorithm finds an error state in cycle 14 of S1423 as can be seen in Figure 5. In this case, finding the error state On-The-Fly is essential, since we were not able to finish reachability on this example even using our parallel system.

Figure 7 demonstrates an incompleteness of our implementation which we did not make it to repair by the submission deadline. In this example, a step backwards from a single state using the original transition relation may result in a very large set which includes a lot of unreachable states. Thus, the processes which are busy in generating the counter example require a lot more memory than the others, resulting in a large difference of the maximum and the average memory utilization. The method described in Section 3.3 ensures that this will not happen by intersecting each partition of the transition relation with the local doughnuts during the backward step.

4.2 Parallel On-the-Fly Model Checking – Timing and Communication

Figure 9 gives the timing breakdown for On-The-Fly model checking on our benchmark suit. The parallel reachability stage takes most of the computation time. As shown in 11, communication does not become a bottleneck at that stage.

5 Conclusions

Large clusters of computers are readily available nowadays. We believe that these environments should be regarded as huge memory pools that can be harnessed for solving joint goals. Our methods can be seen as a globalization of memory systems, using the network as an intermediate level of the memory hierarchy. This intermediate level resides between the main memory and the disk: on the one hand it is a lot faster than the disk (the difference in latency and bandwidth is between three to five orders of magnitude); on the other hand, it is a lot slower than the main memory module (about three orders of magnitude for very fast networks), so locality is still an important issue.

To the best of our knowledge this is the first study reporting on parallel on-the-fly symbolic model checking. Our positive results clearly indicate that this is a promising direction that deserves more attention.

It is important to note that our method was integrated into a high-performance model checker, thus proving its industrial potential. This fact also shows that our parallelization method is orthogonal in many respects to other important optimizations, and does not hurt their applicability.

Acknowledgments

We would like to thank the Formal Method group at IBM Haifa, and specifically Sharon Kidar and Yoav Rodeh for helping us engaging our algorithm with the IBM model checker RuleBase.

References

1. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An Industry-Oriented Formal Verification Tool. In *33rd Design Automation Conference*, pages 655–660, 1996.
2. I. Beer, S. Ben-David, and A. Landver. On-The-Fly Model Checking of RCTL Formulas. In *Proc. of the 10th International Conference on Computer Aided Verification, LNCS 818*, pages 184–194. Springer-Verlag, June-July 1998.
3. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-The-Fly Model Checking for CTL*. In *Proc. of the Conference on Logic in Computer Science (LICS'95)*, June 1995.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991. Winner of the Sidney Michaelson Best Paper Award.

6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
7. Gianpiero Cabodi, Paolo Camurati, and Stefano Que. Improved Reachability Analysis of Large FSM. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
8. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32rd Design Automation Conference*, pages 655–660, 1995.
9. Olivier Coudert, Jean C. Madre, and Christian Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In R. Kurshan and E. M. Clarke, editors, *Workshop on Computer Aided Verification, DIMACS, LNCS 531*, pages 23–32. Springer-Verlag, New Brunswick, NJ, June 1990.
10. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
11. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. of the 12th International Conference on Computer Aided Verification*. Springer-Verlag, June 2000.
12. J.E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley Pub. Co, 1979.
13. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.
14. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
15. Adrian A. Narayan, Jain J. Jawahar Isles, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
16. Adrian A. Narayan, Jain Jawahar, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
17. Doron Peled. Combining Partial Order Reductions with On-The-Fly Model Checking. In *Proc. of the Sixth International Conference on Computer Aided Verification, LNCS 818*, pages 377–390. Springer-Verlag, June 1994.
18. Ulrich Stern and David L. Dill. Parallelizing the Murphy Verifier. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS 1254*, pages 256–267. Springer-Verlag, June 1997.

A Regular Expressions in Symbolic Model-Checking

When specifying a formula in temporal-logic, one describes what should *hold* in the model. Another way to specify a property, is to describe what should *never hold* in the model, thus describing the set of BAD computations rather than the good computations. A nice way to describe a set of finite bad computations is by using *Regular Expressions*(RE), in a special way, as described in the following. Let W be a finite set of symbols (in our case: signal names of the model under test). The alphabet Σ over which the regular expressions are expressed, is the set of all Boolean expressions over W .

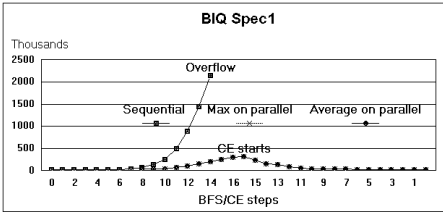
Circuit	#vars		max store size		peak		spec check			gc	
	model	sat	size	step	size	step	time	steps	CE	time	
BIQ											
$SPEC1 : \{[*], (writePtr(0..3) = place(0..3)) \& (dataIn(0) = D(0)),$ $goto((readPtr(0..3) = place(0..3)) Complete)\}$ $(AX[4](DataOut(0) = D(0)))$											
$SPEC2 : \{[*], (writePtr(0..3) = place(0..3)) \& (dataIn(0) = D(0)),$ $goto((readPtr(0..3) = place(0..3)) Complete)\}$ $(AX[2](DataOut(0) = D(0)))$											
SPEC 1	102	5	2,145,201	14	5,852,485	15	15,059	Ov(15)		1,816	
SPEC 2	102	5	2,145,201	14	5,332,126	14	3,811		15	95	1,172
s1423											
$SPEC1 : AG(G729 \& G726 \rightarrow AX[10](G726))$											
$SPEC2 : AG(G729 \& G726 \rightarrow AX[7](G726))$											
SPEC 1	91	4	2,574,709	11	8,640,069	12	2,024	Ov(12)		366	
SPEC 2	91	3	914,046	10	1,540,999	10	625		11	58	132
GXI											
SPEC 1: $\{[*], PACKET-START0 \& ADDR-DATA(0..2)=slot,$ $true, RESP-DATA-IN(0..3)=0010B\}$ $(ABF[2..32](PACKET-START0 \& ADDR-DATA(0..2)=slot))$											
SPEC 1	292	6	3,880,164	43	8,141,305	44	16,222	Ov(44)		3,258	
s5378											
$SPEC1 : AG(n3104gat \rightarrow AX[6]((n3106gat)before(n3104gat)))$											
SPEC 1	188	4	3,914,409	5	9,663,200	6	4,440	Ov(6)		679	

Fig. 2. #vars gives the number of variables in the model and the sat(ellite). All sizes are given in BDD nodes, and all times in seconds. Let reachable be the set of nodes already reached, new – the set of nodes reached but not yet developed, doughnuts – the list of new in preceding steps. Then max store size is the maximal (over the steps) of (reachable + new + doughnuts). The peak is the maximal size at any point during a step. In order to mask the effect of garbage collection (gc) scheduling decisions, the peak is measured after gc invocations. Spec check is the number of steps/time it takes to find an error state, and the time it takes to generate a Counter Example (CE). Ov(*x*) means memory overflow during step *x*. All measurements were done using an RS6000 machine, consisting of a 225MHz PowerPC processor with 512MB memory.

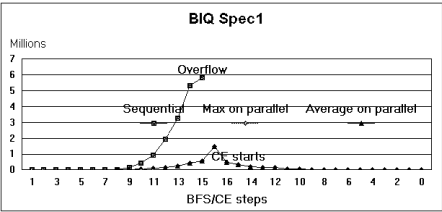
As an example, consider a model with two signals: *req* and *ack*, and consider a property specifying that every *req* must be followed by an *ack* in the next cycle. Σ in this case consists of all 16 possible Boolean functions (*true*, *false*, *req*, $\neg req$, *req* \vee *ack* etc.). Describing the bad computations of this property, we say that sequences with *req* holding in one state and *ack* not holding in the next state, are illegal.

Using Regular expressions we get the following:

$$(true^*)(req)(\neg ack)$$

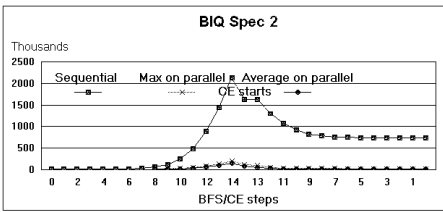


(a) Size of store

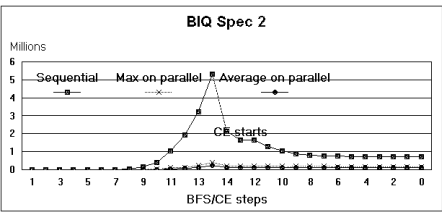


(b) Nodes allocated (peak)

Fig. 3. Memory utilization during On-The-Fly model checking of BIQ spec 1

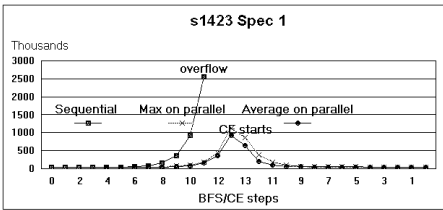


(a) Size of store

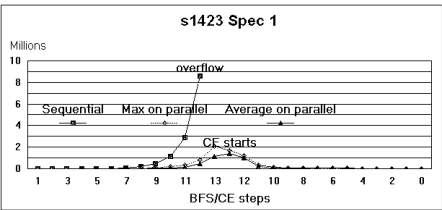


(b) Nodes allocated (peak)

Fig. 4. Memory utilization during On-The-Fly model checking of BIQ, spec 2

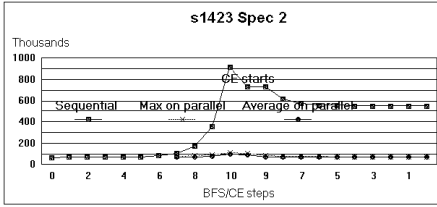


(a) Size of store

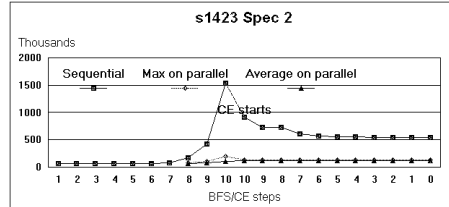


(b) Nodes allocated (peak)

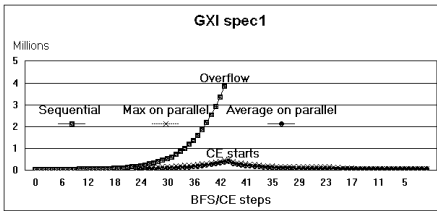
Fig. 5. Memory utilization during On-The-Fly model checking of s1423, spec1



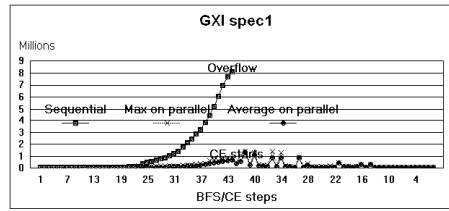
(a) Size of store



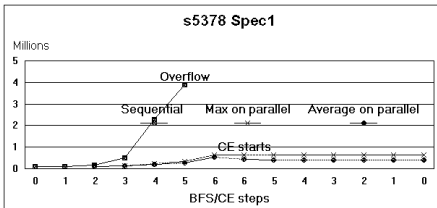
(b) Nodes allocated (peak)

Fig. 6. Memory utilization during On-The-Fly model checking of s1423, spec2

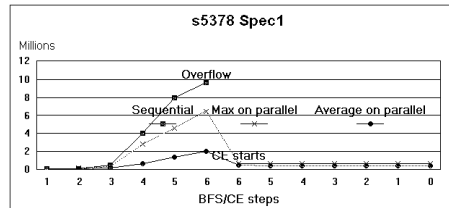
(a) Size of store



(b) Nodes allocated (peak)

Fig. 7. Memory utilization during On-The-Fly model checking of GXI, spec 1

(a) Size of store



(b) Nodes allocated (peak)

Fig. 8. Memory utilization during model checking of s5378, spec 1

Circuit	SPEC	steps	total	Reachability		Spec check		gc
				seq	par	eval	CE	
BIQ		1 17(15)	1,957	174	1,804	31	74	159
		2 15	921	184	731	24	52	88
s1423		1 14(12)	16,032	13	15,911	35	117	1,950
		2 11	521	116	337	10	102	52
GX1		1 45(44)	10,268	1,866	6,570	50	1,738	447
		1 7(6)	12,873	384	11,509	69	105	903

Fig. 9. Timing data (seconds) for parallel execution on 32×512MB machines. Each of the measures is the worst sample over all the machines. The steps count shows that the parallel system always gets farther from where a single-machine overflows (given in brackets). Total is the total time over all steps, including the sequential stage, parallel reachability stage, counter example generation, and garbage collection (gc) time. Note that the total time is the maxima over sums and not the sum over maxima. Seq(ue)ntial is the time it took to get to the threshold at which point the parallel stage started. Par(allel) is the parallel reachability analysis time. Eval(uation) is the total time to evaluate at each step whether one of the processes found an error state (Note that in the sequential stage Eval is a single BDD operation, while in the parallel stage it also requires global interaction over the network). CE is the time to generate the counter example.

In order to check a given model M against a RE specification r , one has to build the corresponding automaton A_r [12], and check that $L(M \cap A_r) = \emptyset$.

Using SMV, we perform this check in the following way: First, we translate A_r into a corresponding non-deterministic finite state-machine F_r in the input language of SMV, with accepting states q_1, \dots, q_n . We then model-check the CTL formula

$$AG(F_r \neq q_1 \wedge \dots \wedge F_r \neq q_n)$$

against the model $M \times F_r$.

Note that the formula to check is of the form $AG(p)$, where p is a boolean formula, and thus can be checked On-The-Fly [132], saving a lot of time and space. Therefore model-checking of Regular Expressions is more efficient in most cases than model-checking of CTL formulas.

Regular Expressions as described above, have a different expressive power than temporal logics. Beer et al in [2] discuss the relations between RE and CTL, and give an algorithm to translate a subset of CTL formulas to Regular Expression specifications. The subset of CTL which can be translated to RE is called *RCTL*.

The Semantics of Verilog Using Transition System Combinators

Gordon J. Pace

Chalmers University of Technology, Sweden
gpace@cs.chalmers.se

Abstract. Since the advent of model checking it is becoming more common for languages to be given a semantics in terms of transition systems. Such semantics allow to model check properties of programs but are usually difficult to formally reason about, and thus do not provide a sufficiently abstract description of the semantics of a language. We present a set of transition system combinators that allow abstract and compositional means of expressing language semantics. These combinators are then used to express the semantics of a subset of the Verilog hardware description language. This approach allows reasoning about the language using both model checking and standard theorem proving techniques.

1 Introduction

Various benefits can be gained through the formal definition of the semantics of a language: documentation of the language semantics, enabling of formal reasoning, and automatic machine reasoning using techniques which allow (relatively) efficient automatic property checking. Giving a semantics through which we benefit from all these is usually difficult to achieve. For example, documenting the full semantics of an industrial strength language tends to yield inelegant semantics that are difficult to reason about, while aiming at automatic machine reasoning one tends to construct semantics which are impractical for interactive formal reasoning which may be necessary to verify large systems.

Hardware description languages, such as Verilog and VHDL have a very complex semantics defined in terms of their simulation behaviour. A lot of work has been done on the formalisation of these languages but most work manages to satisfactorily address only one of the desirable aspects of a formal semantics as listed earlier. Commercial tools for formal verification work only on register transfer level (RTL) descriptions. Furthermore, the semantics they use is usually not formally documented. In this paper, we identify a formal domain which allows us to document the semantics of these languages reasonably elegantly without sacrificing formal reasoning or model checking.

We are particularly interested in automatic machine verification through the use of model checking using BDDs [1], or SAT based techniques [2, 3]. The formal domain we use is a variant on standard transition systems since a wide variety

of techniques have been developed to check properties of transition systems efficiently and automatically. A number of transition system combinators permit us to express the semantics of languages compositionally and also allow effective reasoning about programs in the language.

To illustrate the effectiveness of this approach, we present the semantics of a subset of Verilog [10], which we call VeriSmall. The techniques used for VeriSmall readily scale up for larger subsets of Verilog. A more complete subset of behavioural Verilog has been formalised and implemented in a translator into SMV [6], details of which can be obtained from the author.

2 Notation

A relation between sets A and B is a set of pairs (a, b) where $a \in A$ and $b \in B$. The type of such a relation will be written as $A \leftrightarrow B$. If $\overset{A, B}{\mapsto}$ represents a relation of type $A \leftrightarrow B$, then $\overset{A, B}{\mapsto}_1 \cup \overset{A, B}{\mapsto}_2$ is the union of the two relations. The forward composition of the two relations will be written as $\overset{A, B}{\mapsto}; \overset{B, C}{\mapsto} . \overset{A, B}{\mapsto}^{-1}$ is the inverse of the relation — with type $B \leftrightarrow A$. The relational image of C ($C \subseteq A$) under $\overset{A, B}{\mapsto}$ is written as $\overset{A, B}{\mapsto}(C)$. The restriction of the domain of $\overset{A, B}{\mapsto}$ to set C is written as $\overset{A, B}{\mapsto} \upharpoonright C$ and similarly, $\overset{A, B}{\mapsto} \not\downharpoonright C$ is the restriction of the relation to the complement of C .

The overriding of $\overset{A, B}{\mapsto}_1$ by $\overset{A, B}{\mapsto}_2$, written as $\overset{A, B}{\mapsto}_1 \oplus \overset{A, B}{\mapsto}_2$ relates a with b if, either $a \overset{A, B}{\mapsto}_2 b$ or a is not in the domain of $\overset{A, B}{\mapsto}_2$ and $a \overset{A, B}{\mapsto}_1 b$. We make a relation total by adding any necessary identity transitions: $(\overset{A, A}{\mapsto})^{id} \stackrel{\text{def}}{=} id \oplus \overset{A, A}{\mapsto}$. Finally, $(\overset{A, B}{\mapsto}, \overset{C, D}{\mapsto})$ is the pairwise joining of inputs and outputs of the relations to get a relation of type $(A \times C) \leftrightarrow (B \times D)$.

3 VeriSmall

Verilog is a large and complex language. Documenting the detailed semantics of the whole language is beyond the scope of this paper. We thus work with a syntactic subset of Verilog, which we call VeriSmall. The sub-language is chosen such that the complexities of Verilog semantics are exposed. It is, however, sufficiently small to be presented in its entirety and reasoned about in this paper. The syntax of VeriSmall is given in figure 1.

VeriSmall is a concurrent language with steps made along parallel threads non-deterministically. The `#0` construct (read *zero delay*) blocks a thread until all other threads are finished or are similarly blocked. The behaviour of the rest of the language should be intuitively clear.

To illustrate the effect of zero delays, consider the program `(initial v = 1 || initial v = 0)`. Note that this program is non-deterministic and v can finish with either value 1 or 0. On the other hand, `(initial v = 1 || initial begin`

```

⟨program⟩ ::= ⟨module⟩
           | ⟨program⟩ || ⟨program⟩
⟨module⟩  ::= initial ⟨statement⟩
           | always ⟨statement⟩
⟨statement⟩ ::= skip
            | ⟨variable⟩ = ⟨expression⟩
            | wait ( ⟨variable⟩ )
            | #0 ⟨statement⟩
            | while ( ⟨expression⟩ ) ⟨statement⟩
            | if ( ⟨expression⟩ ) ⟨statement⟩ else ⟨statement⟩
            | begin ⟨block⟩ end
⟨block⟩   ::= ⟨statement⟩
           | ⟨block⟩ ; ⟨block⟩

```

Fig. 1. The syntax of VeriSmall

`#0 v = 0 end`) is a deterministic program and always terminates with v carrying value 0.

Despite the fact that we have stripped Verilog down to its bare essentials, VeriSmall is still a substantially complex language.

3.1 The Simulation Cycle

The official documentation of Verilog describes the behaviour of a program in terms of how it should be interpreted by a simulator. This is also how we will informally describe the behaviour of VeriSmall programs.

VeriSmall is a concurrent language with a scheduling algorithm which manages the order of execution of concurrent threads. Each thread can be in one of three modes: **enabled**, **delayed(0)**, **waitfor(v)** or **finished**. If a state is in mode **waitfor(v)** and the value of v is high, the thread is put into **enabled** mode. If no such threads exist, then the scheduler executes an **enabled** thread and the process is repeated. When no **enabled** threads are available, all threads in **delayed(0)** mode are made **enabled** and the process repeated from the beginning. The values of variables are also stored by the simulator. In VeriSmall, variables range over the values 1 (high), 0 (low), z (high impedance) and \times (unknown). The simulation cycle is shown in figure 1.

The actions performed when moving along a thread depend on the instruction currently pointed at by the thread pointer.

Skip: The thread pointer is moved forward.

Assignments: If the first instruction is an assignment $v = e$, then the expression e is evaluated in the current state and v set to the value calculated.

Zero Delays: To move along a thread pointing at `#0 P` we set the thread's state to **delayed(0)** and move the thread pointer to P .

Conditionals: To move along the statement `if (e) P else Q`, e is evaluated and depending on its value, the thread pointer is moved to point at P or Q .

```

initialise all variables to ×
set all thread states to enabled
forever do
  if (there are threads waiting on a true variable) then
    set them enabled
  elsif (there are enabled threads) then
    choose one non-deterministically
    move one step along the thread
  elsif (any delayed by 0 modules) then
    enable all such modules
end

```

Fig. 2. VeriSmall simulation cycle

Wait: If the statement is `wait(v)` then the thread's state is set to `waitfor(v)` and the thread pointer is moved forward.

Loops: To move along the statement `while (e) P`, *e* is evaluated and depending on its value, the thread pointer is set to point at *P* or the first instruction after the loop instruction.

Blocks: The instruction pointer is moved to the first instruction in the block.

The top level module instructions are simply syntactic sugar, with `always P` corresponding to `while (1) P` and `initial P` to `P`.

4 Layered Transition Systems

Hardware description languages, the languages we are mainly interested in expressing the semantics of, usually have an inherent concept of priority of execution. In the case of VeriSmall, for instance, threads blocked by a zero delay get lower priority than enabled ones.

With this specification in mind, we add extra priority information by placing all states in one of an ordered set of layers — the higher the the layer, the higher priority given to that state when composing systems in parallel. We also use standard transition system combinators such as union and sequential composition, as intermediate language constructs.

It is important to note that the layering information is only necessary to compose systems and can be hidden away when we want to check properties of a transition system.

4.1 Formal Definition

Since we would like to be able to compare the layering information in one machine to that in another, we will assume the existence of a fixed set of layers *LAYER* over which the total ordering \geq is defined.

Definition 1: A *finite layered transition system* (henceforth FLTS), is a 5-tuple $\langle Q, I, F, \mapsto, \text{layer} \rangle$, where:

- Q finite set of states,
- I set of initial states ($I \subseteq Q$),
- F set of final states ($F \subseteq Q$),
- \mapsto transition relation between states
- layer total function from states to layers

We will use \mathcal{M} to represent arbitrary FLTS. If necessary, we will also use subscripts. Unless otherwise stated, \mathcal{M} is the FLTS $\langle Q, I, F, \mapsto, \text{layer} \rangle$ and \mathcal{M}_i is $\langle Q_i, I_i, F_i, \mapsto^i, \text{layer}_i \rangle$.

4.2 Transition System Combinators

Definition 2: Given a FLTS \mathcal{M} and a predicate $r_i : Q \rightarrow \text{bool}$, then \mathcal{M} *domain restricted to r_i* , is defined by:

$$r_i \triangleleft \mathcal{M} \stackrel{\text{def}}{=} \langle Q, \{\sigma : I \mid r_i(\sigma)\}, F, \mapsto, \text{layer} \rangle$$

Definition 3: Given a FLTS \mathcal{M} and a predicate $r_f : Q \rightarrow \text{bool}$, then \mathcal{M} *range restricted to r_i* , is defined by:

$$\mathcal{M} \triangleright r_f \stackrel{\text{def}}{=} \langle Q, I, \{\sigma : F \mid r_f(\sigma)\}, \mapsto, \text{layer} \rangle$$

Definition 4: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 , we define the *union* of the two systems $\mathcal{M}_1 \cup \mathcal{M}_2$ as follows:

$$\mathcal{M}_1 \cup \mathcal{M}_2 \stackrel{\text{def}}{=} \langle Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \mapsto^1 \cup \mapsto^2, \text{layer}_1 \cup \text{layer}_2 \rangle$$

Note that the two transition systems *must* agree on the layer of any common states.

Definition 5: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 and a relation jn between the final states of \mathcal{M}_2 and the initial of \mathcal{M}_1 ($jn : F_1 \leftrightarrow I_2$), we define the *catenation* of the two systems $\mathcal{M}_1 \stackrel{jn}{;} \mathcal{M}_2$:

$$\mathcal{M}_1 \stackrel{jn}{;} \mathcal{M}_2 \stackrel{\text{def}}{=} \langle Q_1 \cup Q_2, jn^{id}(I_1), F_2, \mapsto^1; jn^{id} \cup \mapsto^2, \text{layer}_1 \cup \text{layer}_2 \rangle$$

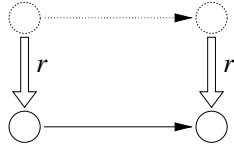
Thus, whenever a transition can take us from a state σ to state σ' which is related (via jn) to σ'' , we add the direct transition from σ to σ'' . States in \mathcal{M}_2 related to initial states in \mathcal{M}_1 are also initial.

Since we have catenation, we can similarly define the reflexive transitive closure of a FLTS.

Definition 6: Given a FLTS \mathcal{M} and a relation jn between the final states and initial states ($jn : F \leftrightarrow I$, with disjoint domain and range) we define the *reflexive transitive closure* of \mathcal{M} — written as \mathcal{M}^* — as follows:

$$\mathcal{M}^* \stackrel{\text{def}}{=} \langle Q, jn^{id}(I), F \setminus \text{dom}(jn), \mapsto; jn^{id}, \text{layer} \rangle$$

Given a FLTS we sometimes need to re-map the states — this can be seen as a form of data abstraction. Given a relation r between the old and the new states, we can use this relation to create a new FLTS such that there is a transition between two new states σ and σ' if and only if they are related (by r) to two old states between which there was a transition.



Definition 7: Given a FLTS \mathcal{M} and relation $r : Q \leftrightarrow Q'$ we can define the *remapping of \mathcal{M} with r* , written as $r(\mathcal{M})$, as follows:

$$r(\mathcal{M}) \stackrel{\text{def}}{=} \langle Q', r(I), r(F), r^{-1}; \mapsto; r, \lambda\sigma \cdot \max(r^{-1}; \text{layer}(\sigma)) \rangle$$

Note that in some cases, r may relate several old states with a new one. In this case the new state assumes the highest priority of its related states.

4.3 Parallel Composition

To compose two systems in parallel we need to define what the result of joining two states is. It is tempting to always take the Cartesian product of the old state spaces to be the new state space. However, this will complicate matters when we have states with overlapping domains. We thus choose to explicitly express this operation as two relations \succ_1 and \succ_2 which join two states together giving priority to the first and second state respectively.

$$\succ_1, \succ_2: (Q_1 \times Q_2) \rightarrow Q$$

When we do not care which state is given priority, we will use \succ defined as $\succ_1 \cup \succ_2$.

We will also need to decompose states into separate parts:

$$\prec: Q \rightarrow (Q_1 \times Q_2)$$

Note that all the following FLTS combinators are parametrised by these functions.

Given two FLTS, we can construct a FLTS acting like the coupling of the two FLTS — this corresponds to running the two systems together, where for every transition one system makes, the other also performs exactly one transition.

Definition 8: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 , we can define the *synchronous parallel composition* of the two systems $\mathcal{M}_1 \parallel \mathcal{M}_2$ as follows:

$$\begin{aligned} \mathcal{M}_1 \parallel \mathcal{M}_2 \stackrel{\text{def}}{=} & \langle \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (\overset{1}{\mapsto}, \overset{2}{\mapsto}); \succ, \prec; (\text{layer}_1, \text{layer}_2); \max_2 \rangle \end{aligned}$$

(where \max_2 relates a pair of numbers with the maximum of the two)

We will write this transition relation as $(\overset{1}{\mapsto} \parallel \overset{2}{\mapsto})$.

Sometimes it is necessary to allow the processes to stutter — while one of the processes carries out a transition, the other remains in the same state.

Definition 9: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 , we define the *parallel composition with stuttering* of the two systems $\mathcal{M}_1 \parallel^s \mathcal{M}_2$ as follows:

$$\begin{aligned} \mathcal{M}_1 \parallel^s \mathcal{M}_2 \stackrel{\text{def}}{=} & \langle \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (\overset{1}{\mapsto} \cup id, \overset{2}{\mapsto}); \succ_2 \cup \prec; (\overset{1}{\mapsto}, \overset{2}{\mapsto} \cup id); \succ_1, \\ & \prec; (\text{layer}_1, \text{layer}_2); \max_2 \rangle \end{aligned}$$

We will write this composition of transition relations as $(\stackrel{1}{\mapsto} \parallel^\iota \stackrel{2}{\mapsto})$.

Sometimes we would like to compose two systems concurrently, with only one system performing a transition at a time.

Definition 10: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 , we define the *interleaving* of the two systems $\mathcal{M}_1 \parallel \mathcal{M}_2$ as follows:

$$\begin{aligned} \mathcal{M}_1 \parallel \mathcal{M}_2 \stackrel{\text{def}}{=} & \langle \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (id, \stackrel{2}{\mapsto}); \succ_2 \cup \prec; (\stackrel{1}{\mapsto}, id); \succ_1, \\ & \prec; (layer_1, layer_2); \max_2 \rangle \end{aligned}$$

We will write this composed transition relation as $(\stackrel{1}{\mapsto} \parallel \stackrel{2}{\mapsto})$.

4.4 Layered Parallel Composition

Definition 11: Given two FLTS \mathcal{M}_1 and \mathcal{M}_2 and a function which determines how the layers will be composed: $+: LAYER \rightarrow \{\parallel, \parallel^\iota, \parallel\}$, we define the *layered parallel composition* of the two systems $\mathcal{M}_1 + \mathcal{M}_2$ as follows:

$$\begin{aligned} \mathcal{M}_1 + \mathcal{M}_2 \stackrel{\text{def}}{=} & \langle \succ (S_1 \times S_2), \\ & \succ (I_1 \times I_2), \succ (F_1 \times F_2), \mapsto, \\ & \prec; (layer_1, layer_2); \max_2 \rangle \end{aligned}$$

where

$$\begin{aligned} \sigma \mapsto \sigma' \stackrel{\text{def}}{=} & \text{layer}(\prec \sigma)_1 > \text{layer}(\prec \sigma)_2 \wedge \sigma(\stackrel{1}{\mapsto} \parallel \emptyset) \sigma' \vee \\ & \text{layer}(\prec \sigma)_1 < \text{layer}(\prec \sigma)_2 \wedge \sigma(\emptyset \parallel \stackrel{2}{\mapsto}) \sigma' \vee \\ & \text{layer}(\prec \sigma)_1 = \text{layer}(\prec \sigma)_2 = i \wedge \sigma(\stackrel{1}{\mapsto} +_i \stackrel{2}{\mapsto}) \sigma' \end{aligned}$$

This means that if one of the state components has a higher priority, the other component is not allowed to perform any transitions (lines 1 and 2) while, if both components have the same priority, then the transition relation is determined by the layer in which they lie (line 3).

4.5 FLTS Combinator Expressions

Definition 12: We define TS_{\equiv} to be the language of all valid combinator expressions over layered transition systems with basic transition systems as terminals.

TS_{\equiv}^V is the similar language but which may also have variables (elements of V) as terminals.

A *context* $C(X)$ is an expression in $TS_{\equiv}^{\{X\}}$. Given an expression $\mathcal{M} \in TS_{\equiv}$, $C(\mathcal{M})$ is the same as $C(X)$, but with instances of X replaced by \mathcal{M} .

5 Laws of FLTS

The FLTS combinators presented in the previous section allow us to specify the semantics of VeriSmall and similar languages in a concise and readable manner. They also satisfy a number of laws which will enable us to reason about VeriSmall programs more easily. In this section we present a number of such laws.

5.1 Reachability and Inclusion

In this paper the only semantic property of FLTS we are concerned with is reachability:

Definition 13: Given a FLTS \mathcal{M} , we define the set of reachable states from states Σ as follows:

$$\text{reachable}_{\mathcal{M}}(\Sigma) \stackrel{\text{def}}{=} \mapsto^* (\Sigma)$$

If we only care about the reachable states in a particular FLTS (starting from the initial states), we write this as:

$$\text{reachable}(\mathcal{M}) \stackrel{\text{def}}{=} \text{reachable}_{\mathcal{M}}(I)$$

Given a projection function $\pi : (Q_1 \cup Q_2) \rightarrow Q$, we say that σ and σ' are π -equivalent if $\pi(\sigma) = \pi(\sigma')$. We write this as $\sigma =_{\pi} \sigma'$. Similarly, we can extend this notation to sets of states:

$$\Sigma \subseteq_{\pi} \Sigma' \stackrel{\text{def}}{=} \pi(\Sigma) \subseteq \pi(\Sigma')$$

We will define FLTS containment with respect to a transition function.

Definition 14: A FLTS \mathcal{M}_1 is said to be *contained in* \mathcal{M}_2 *with respect to a projection* π , written as $\mathcal{M}_1 \sqsubseteq_{\pi} \mathcal{M}_2$, if:

- States are contained under π : $Q_1 \subseteq_{\pi} Q_2$
- Initial states are contained under π : $I_1 \subseteq_{\pi} I_2$
- Final states are inversely contained under π : $F_2 \subseteq_{\pi} F_1$
- π -equivalent elements in Q_1 and in Q_2 belong to the same layer:
 $\forall \sigma : Q_1, \sigma' : Q_1.$

$$\sigma =_{\pi} \sigma' \implies \text{layer}_1(\sigma) = \text{layer}_2(\sigma')$$

This is equivalent to: $\pi^{-1}; \text{layer}_1 = \pi^{-1}; \text{layer}_2$

- A member of Q_2 must be able to emulate (up to π) all transitions of π -equivalent members of Q_1 :

$$\forall (\sigma_1, \sigma'_1) : \mapsto^1, \sigma_2 : Q_2.$$

$$\sigma_1 =_{\pi} \sigma_2 \implies \exists \sigma'_2 : Q_2 \cdot \sigma_2 \mapsto^2 \sigma'_2 \wedge \sigma'_1 =_{\pi} \sigma'_2$$

This is equivalent to: $\pi; \pi^{-1}; \mapsto^1 \subseteq \mapsto^2; \pi; \pi^{-1}$

Transition system inclusions guarantee containment of reachable states.

Lemma 1: If $\mathcal{M}_1 \sqsubseteq_{\pi} \mathcal{M}_2$ then, for any set of states Σ , $\text{reachable}_{\mathcal{M}_1}(\Sigma) \subseteq_{\pi} \text{reachable}_{\mathcal{M}_2}(\Sigma)$.

5.2 Monotonicity

Provided that π obeys a number of properties (dictating how it distributes over the relations used in FLTS construction) expressions in TS_{\equiv} are monotone with respect to \sqsubseteq_{π} . These proofs follow almost exclusively from the monotonicity of relational mapping, relational composition and set union. A frequently used property of functions is $\pi; \pi^{-1}; \pi = \pi$.

Lemma 2: If $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_3$ and $\mathcal{M}_2 \sqsubseteq_\pi \mathcal{M}_4$ then:

- i) Reduction of initial states: If $i = \pi; i$ then $i \triangleleft \mathcal{M}_1 \sqsubseteq_\pi i \triangleleft \mathcal{M}_3$.
- ii) Reduction of final states: If $f = \pi; f$ then $\mathcal{M}_1 \triangleright f \sqsubseteq_\pi \mathcal{M}_3 \triangleright f$.
- iii) Remapping: If $r; \pi; \pi^{-1} = \pi; \pi^{-1}; r$ then $r(\mathcal{M}_1) \sqsubseteq_\pi r(\mathcal{M}_3)$.
- iv) Union: $\mathcal{M}_1 \cup \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \cup \mathcal{M}_4$.
- v) Sequential composition: If $jn^{id}; \pi; \pi^{-1} = \pi; \pi^{-1}; jn^{id}$ then $\mathcal{M}_1 \stackrel{jn}{;} \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \stackrel{jn}{;} \mathcal{M}_4$.
- vi) Reflexive transitive closure: If $jn^{id}; \pi; \pi^{-1} = \pi; \pi^{-1}; jn^{id}$ and if also $\sigma =_\pi \sigma' \implies (\sigma \in \text{dom } jn \Leftrightarrow \sigma' \in \text{dom } jn)$, then $\mathcal{M}_1^* \sqsubseteq_\pi \mathcal{M}_2^*$.
- vii) Synchronous parallel composition: If the relation $\pi; \pi^{-1}$ commutes with state composition and state decomposition:

$$\begin{aligned} \succ; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ \\ \prec; (\pi; \pi^{-1}, \pi; \pi^{-1}) &= \pi; \pi^{-1}; \prec \end{aligned}$$
 then $\mathcal{M}_1 \parallel \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \parallel \mathcal{M}_4$.
- viii) Parallel composition with stuttering: Similarly, if $\pi; \pi^{-1}$ commutes with state composition and state decomposition:

$$\begin{aligned} \succ_1; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ_1 \\ \succ_2; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ_2 \\ \pi; \pi^{-1}; \prec &= \prec; (\pi; \pi^{-1}, \pi; \pi^{-1}) \end{aligned}$$
 then $\mathcal{M}_1 \parallel^t \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \parallel^t \mathcal{M}_4$.
- ix) Interleaving: Under the same constraints as case viii, $\mathcal{M}_1 \parallel \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \parallel \mathcal{M}_4$.
- x) Layered parallel composition: Under the same constraints as case viii, $\mathcal{M}_1 + \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 + \mathcal{M}_4$.

Proof: The proofs of the different cases are very similar and thus, we just present the proof of case ix.

States:

$\pi(\text{states of } \mathcal{M}_1 \parallel \mathcal{M}_3)$
 $= \{ \text{definition of interleaving} \}$
 $\succ; \pi(Q_1 \times Q_3)$
 $= \{ \pi = \pi; \pi^{-1}; \pi \}$
 $\succ; \pi; \pi^{-1}; \pi(Q_1 \times Q_3)$
 $= \{ \text{assumption about } \pi \text{ and } \succ \}$
 $(\pi; \pi^{-1}, \pi; \pi^{-1}); \succ; \pi(Q_1 \times Q_3)$
 $\subseteq \{ \mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_3 \text{ and } \mathcal{M}_2 \sqsubseteq_\pi \mathcal{M}_4 \}$
 $(\pi; \pi^{-1}, \pi; \pi^{-1}); \succ; (\pi(Q_2) \times \pi(Q_4))$
 $= \{ \text{refold back} \}$
 $\succ; \pi(Q_2 \times Q_4)$
 $= \{ \text{definition of interleaving} \}$
 $\pi(\text{states of } \mathcal{M}_2 \parallel \mathcal{M}_4)$

Transitions:

$\pi; \pi^{-1}; (\text{transition relation of } \mathcal{M}_1 \parallel \mathcal{M}_3)$
 $= \{ \text{definition of interleaving} \}$
 $\pi; \pi^{-1}; \prec; (id, \overset{3}{\mapsto}); \succ_2 \cup$
 $\pi; \pi^{-1}; \prec; (\overset{1}{\mapsto}, id); \succ_1$
 $= \{ \text{assumptions about } \pi \}$
 $\prec; (\pi; \pi^{-1}; id, \pi; \pi^{-1}; \overset{3}{\mapsto}); \succ_2 \cup$
 $\prec; (\pi; \pi^{-1}; \overset{1}{\mapsto}, \pi; \pi^{-1}; id); \succ_1$
 $\subseteq \{ \mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_2 \text{ and } \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_4 \}$
 $\prec; (\pi; \pi^{-1}; id, \overset{4}{\mapsto}; \pi; \pi^{-1}); \succ_2 \cup$
 $\prec; (\overset{2}{\mapsto}; \pi; \pi^{-1}, \pi; \pi^{-1}; id); \succ_1$
 $= \{ \text{refold back} \}$
 $\prec; (id, \overset{4}{\mapsto}); \succ_2; \pi; \pi^{-1} \cup$
 $\prec; (\overset{2}{\mapsto}, id); \succ_1; \pi; \pi^{-1}$
 $= \{ \text{definition of interleaving} \}$
 $(\text{transition relation of } \mathcal{M}_2 \parallel \mathcal{M}_4); \pi; \pi^{-1}$

Initial and final states information and the layer information proofs proceed similar to the ones above.

□

Theorem 1: Combinator contexts are monotonic. If $\mathcal{M}_1 \sqsubseteq_{\pi} \mathcal{M}_2$ then $C(\mathcal{M}_1) \sqsubseteq_{\pi} C(\mathcal{M}_2)$.

Proof: This follows directly using structural induction and lemma 2. \square

6 Formal Semantics of VeriSmall

We now document the formal semantics of VeriSmall in terms of FLTS. Note that the use of layers allows us to avoid having worry about the details of the complex simulation cycle when describing the semantics of sequential programs. The state of the transition system corresponds to the state of the simulator: the values stored by variables, the position in the different threads and the state of each thread.

The state will be a store: a ‘function’ from variable names to values¹. $\sigma(v)$ is the value of variable in state σ . We extend this notation to expressions. Thus, for example, $\sigma(e \text{ and } f)$ will be defined to be $\sigma(e)$ and $\sigma(f)$. Special variable names *pos* and *grd* (are used to represent the position in the thread and its state respectively. σ_{var} is the store restricted to Verilog variables (that is excluding the position and state variables).

$\sigma[v := e]$ is the state just like σ but with the value of variable v changed to $\sigma(e)$. The set of all states with Verilog variables in V , the value of *pos* ranging over P and that of *grd* over G will be referred to as $\Sigma_{V,P,G}$.

In the course of the language semantics definition, we will sometimes need to know the size of a statement or statement block. Given a statement P we can define $\text{size}(P)$ using primitive recursion over the structure of P :

$$\begin{array}{ll}
 \text{size}(\text{skip}) & \stackrel{\text{def}}{=} 1 \\
 \text{size}(v = e) & \stackrel{\text{def}}{=} 1 \\
 \text{size}(\text{wait } (v)) & \stackrel{\text{def}}{=} 1 \\
 \text{size}(\text{if } e \text{ P else } Q) & \stackrel{\text{def}}{=} 1 + \text{size}(P) + \text{size}(Q) \\
 \text{size}(P; Q) & \stackrel{\text{def}}{=} \text{size}(P) + \text{size}(Q) \\
 \text{size}(\text{begin } P \text{ end}) & \stackrel{\text{def}}{=} 1 + \text{size}(P) \\
 \text{size}(\#0 P) & \stackrel{\text{def}}{=} 1 + \text{size}(P) \\
 \text{size}(\text{while } e \text{ P}) & \stackrel{\text{def}}{=} 1 + \text{size}(P)
 \end{array}$$

It can be proved that the value of *pos* in any state of the transition system produced from a sequential program P will never reach or exceed $\text{size}(P)$.

Similarly we can define the function $\text{vars}(x)$ which returns the set of variables occurring in program or expression x .

Layers and Other Preliminaries. The subset of Verilog we present uses four layers: unblocking states waiting on a variable takes highest priority, followed by transitions on enabled threads are of highest priority, then threads delayed by zero time and finally, the finished threads. Thus:

¹ The quotes around the word function are there because different variables in the domain of a store may return values of different types and therefore it is not, strictly speaking, a function.

$$\begin{aligned} LAYER &\stackrel{\text{def}}{=} \{\text{FIRE}, \text{ENA}, \text{DEL}_0, \text{FIN}\} \\ \text{FIRE} &> \text{ENA} > \text{DEL}_0 > \text{FIN} \end{aligned}$$

These layer names will also be overloaded with the related constant function — for example ENA will also be used for $\lambda\sigma \cdot \text{ENA}$.

We will often need to combine together different automata but making sure that the states are disjoint. This will be done by modifying the pos values using a remapping of the FLTS. If pos_{+n} is the function $pos_{+n}(\sigma) = \sigma[pos := pos + n]$, then the remapping $pos_{+n}(\mathcal{M})$ is the renaming we require.

Each FLTS produced will handle a particular set of variables. To increase the set of variables handled by a transition system we use interleaving composition. If S is $\Sigma_{V, \emptyset, \emptyset}$ then

$$add_V(\mathcal{M}) \stackrel{\text{def}}{=} \langle S, S, S, \emptyset, \text{FIN} \rangle \parallel \mathcal{M}$$

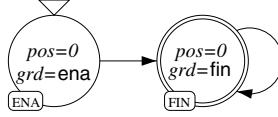
The relations used for this composition are the minimal relations satisfying:

$$\begin{aligned} \sigma &\prec (\sigma \upharpoonright V, \sigma \not\upharpoonright V) \\ (\sigma_1, \sigma_2) &\succ_1 \sigma_2 \oplus \sigma_1 \\ (\sigma_1, \sigma_2) &\succ_2 \sigma_1 \oplus \sigma_2 \end{aligned}$$

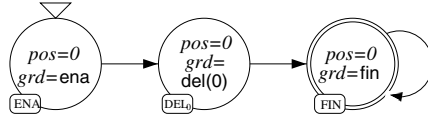
Expressions such as $add_V(pos_{+n}(\mathcal{M}))$ will be written as $\mathcal{M}_{V, +n}$.

The Semantics.

Skip: The interpretation of **skip**, $\llbracket \text{skip} \rrbracket$ is a FLTS with two states:



Zero Delays: Note that all zero delays are followed by a statement. However, this simply corresponds to normal sequential composition and we can give the semantics of zero delays as independent statements and then define $\llbracket \#0 \ P \rrbracket$ to be $\llbracket \#0; P \rrbracket$. The semantics of zero delays $\llbracket \#0 \rrbracket$ is rather similar to that of **skip**:



Assignments: $\llbracket v = e \rrbracket$ is a FLTS with the following set of states:

$$Q = \Sigma_{\text{vars}(v=e), \{0\}, \{\text{enabled}, \text{finished}\}}$$

The initial states are those in which the thread is enabled:

$$\{\sigma : Q \mid \sigma(\text{grd}) = \text{enabled}\}$$

The final states are those in which the thread is finished:

$$\{\sigma : Q \mid \sigma(\text{grd}) = \text{finished}\}$$

The system can go from enabled states to finished ones by setting the value of the variable to that of the expression.

$$\sigma[\text{grd} := \text{enabled}] \mapsto \sigma[v := e][\text{grd} := \text{finished}]$$

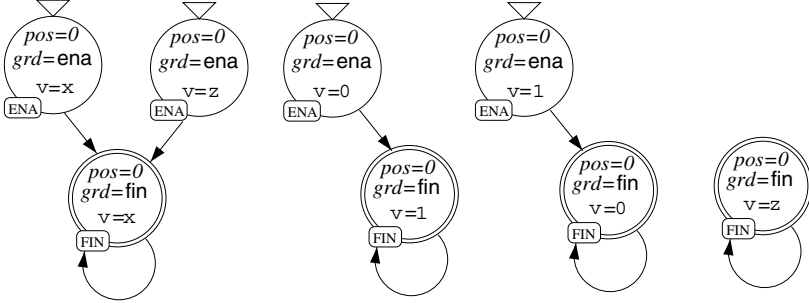
Finished states can perform reflexive transitions:

$$\sigma[grd := finished] \mapsto \sigma[grd := finished]$$

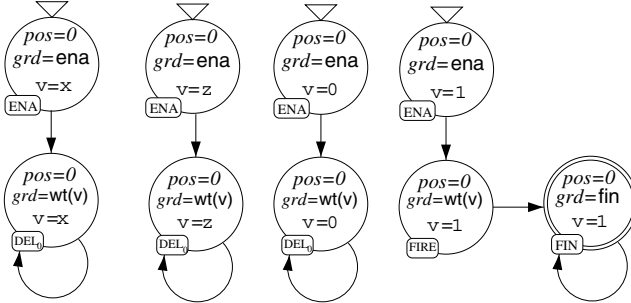
The layer of a state can be deduced from its guard:

$$\text{layer}(\sigma) = \begin{cases} \text{ENA} & \text{if } \sigma(grd) = \text{enabled} \\ \text{FIN} & \text{otherwise} \end{cases}$$

Below is the FLTS $\llbracket v \neq !v \rrbracket$:



Wait: The semantics of $\text{wait}(v)$ are given by the FLTS in the diagram below:



The first step the simulator may perform on such a thread is setting the guard to $\text{waitfor}(v)$ and keep the variable value constant. States guarded by $\text{waitfor}(v)$ and in which v is not high can only remain the same state, but if v is high, the system can proceed to terminate.

Assumptions: To make the presentation of the semantics clearer, we introduce a new statement in VeriSmall:

$$\langle \text{statement} \rangle ::= \langle \text{expression} \rangle^\top$$

e^\top , read *assume e*, moves along the thread if e evaluates to 1 but aborts the thread if not. To complete the definition of the size function: $\text{size}(e^\top) \stackrel{\text{def}}{=} 0$. The set of states is:

$$Q = \{ \sigma : \Sigma_{\text{vars}(e), \{0\}, \{\text{finished}\}} \mid \sigma(e) = 1 \}$$

The semantics of assumption:

$$\llbracket e^\top \rrbracket \stackrel{\text{def}}{=} \langle Q, Q, Q, id, \text{FIN} \rangle$$

Sequential Composition: $\llbracket P; Q \rrbracket$ can be expressed in terms of the FLTS $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$.

$$\llbracket P; Q \rrbracket \stackrel{\text{def}}{=} \llbracket P \rrbracket_{\text{vars}(Q)}^{jn} ; \llbracket Q \rrbracket_{\text{vars}(P), +\text{size}(P)}$$

jn relates states with the same store: $\sigma(jn)\sigma'$ if and only if $\sigma_{var} = \sigma'_{var}$ (and $\sigma \in F_1$ and $\sigma' \in I_2$).

Blocks: It is tempting to define the semantics of a block simply by removing the outer **begin** **end** keywords. Note, however, that the simulator takes one cycle to proceed to the first instruction. An accurate description of the semantics is thus:

$$\llbracket \text{begin } P \text{ end} \rrbracket \stackrel{\text{def}}{=} \llbracket \text{skip}; P \rrbracket$$

Conditionals: The semantics of conditionals can be expressed using restriction of initial states. However, the semantics are slightly more complex due to the fact that the simulator takes one cycle to evaluate the condition:

$$\begin{aligned} \llbracket \text{if } (e) \ P \ \text{else} \ Q \rrbracket &\stackrel{\text{def}}{=} \\ &\llbracket \text{skip} \rrbracket_V \stackrel{jn}{;} \\ &\left(\begin{array}{l} \llbracket (e = 1)^\top; P \rrbracket_{\text{vars}(Q), +1} \cup \\ \llbracket (e \neq 1)^\top; Q \rrbracket_{\text{vars}(P), +1 + \text{size}(P)} \end{array} \right) \end{aligned}$$

where $V = \text{vars}(\text{if } (e) \ P \ \text{else} \ Q)$ and the jn relation is the same as the one used in sequential composition.

While Loops: The semantics of while loops are similar:

$$\begin{aligned} \llbracket \text{while } (e) \ P \rrbracket &\stackrel{\text{def}}{=} \\ &\llbracket \text{skip} \rrbracket_V \stackrel{jn}{;} \\ &(\llbracket (e = 1)^\top; P; \text{skip} \rrbracket^* \cup \llbracket (e \neq 1)^\top \rrbracket_{\text{vars}(P)}) \end{aligned}$$

Again note that the simulator takes a cycle to evaluate the expression and that the jn relation is the same as the one used in sequential composition.

Parallel Composition: Parallel composition can now be defined in terms of layered composition:

$$\llbracket P \parallel Q \rrbracket \stackrel{\text{def}}{=} \llbracket P \rrbracket + \llbracket Q \rrbracket$$

This layered composition uses:

$$+_{\text{FIRE}} = \parallel \quad +_{\text{ENA}} = \parallel \quad +_{\text{DEL}_0} = \parallel \quad +_{\text{FIN}} = \parallel$$

The state constructor relations are:

$$\begin{aligned} (\sigma, \sigma') &\succ_1 \sigma'_{\text{vars}} \oplus \sigma_{\text{vars}} \oplus \{ \text{pos} \mapsto (\sigma(\text{pos}), \sigma'(\text{pos})), \\ &\quad \text{grd} \mapsto (\sigma(\text{grd}), \sigma'(\text{grd})) \} \\ (\sigma, \sigma') &\succ_2 \sigma_{\text{vars}} \oplus \sigma'_{\text{vars}} \oplus \{ \text{pos} \mapsto (\sigma(\text{pos}), \sigma'(\text{pos})), \\ &\quad \text{grd} \mapsto (\sigma(\text{grd}), \sigma'(\text{grd})) \} \end{aligned}$$

The state decomposition relation is simply the inverse of these two: $\prec \stackrel{\text{def}}{=} \succ^{-1}$.

7 Reasoning about VeriSmall

Finally, we show how these semantics can be used to prove general theorems about programs. These may later be used to aid or simplify automatic verification of programs.

Programs are usually built in different parts which are eventually joined together. It is therefore important to be able to show that individual parts satisfy certain properties whatever is plugged into them. The question thus arises: How can we prove such properties of our programs using model checking?

The scenario depicted in the previous paragraph corresponds to a program being a context with variables pointing out where other programs are to be plugged

in. Thus, for example, one may be given the program context $C(P)$: **initial** **begin** $x=0$; P ; $x=!y$ **end** and be asked to prove that if P is a program using only variable y , then x and y are never high at the same time.

The technique we use is to substitute P with the most non-deterministic program possible and prove the property of this new program. If we can prove that:

$$\sigma \in \text{reachable}(\llbracket C(\text{chaos}(\{y\})) \rrbracket) \implies \sigma(x \neq 1 \vee y \neq 1)$$

then it should follow that for any program P which has alphabet $\{y\}$:

$$\sigma \in \text{reachable}(\llbracket C(P) \rrbracket) \implies \sigma(x \neq 1 \vee y \neq 1)$$

Note that if we define the FLTS semantics of $\text{chaos}(V)$, the first statement can be checked automatically using standard reachability techniques.

7.1 Chaos

The semantics of $\text{chaos}(V)$, where V is a set of variables, is straightforward to define.

The states are: $Q = \Sigma_{V, \{0\}, \{\text{enabled}, \text{delayed}(0), \text{finished}\}}$

The initial states are those enabled: $\{\sigma : Q \mid \sigma(\text{grad}) = \text{enabled}\}$

The final states are those which are finished: $\{\sigma : Q \mid \sigma(\text{grad}) = \text{finished}\}$

States can do anything, but once finished they must remain so:

$$id \cup ((Q \setminus F) \times Q)$$

The layer can be deduced from the mode:

$$\text{layer}(\sigma) = \begin{cases} \text{ENA} & \text{if } \sigma(\text{grad}) = \text{enabled} \\ \text{DEL}_0 & \text{if } \sigma(\text{grad}) = \text{delayed}(0) \\ \text{FIN} & \text{if } \sigma(\text{grad}) = \text{finished} \end{cases}$$

Also, $\text{size}(\text{chaos}(V)) \stackrel{\text{def}}{=} 1$ and $\text{vars}(\text{chaos}(V)) \stackrel{\text{def}}{=} V$

7.2 The Theorem

It is quite easy to formulate the result we desire incorrectly. For example, in the example given earlier, the safety condition that the thread position never exceeds 4 can be proved of $C(\text{chaos}(V))$ but this will not be true for long enough instances of P . It is thus important to restrict safety conditions to variables and guards. Similarly we must make sure that P uses no variables other than those in V .

$$\frac{\begin{array}{l} \text{vars}(P) = V \\ \forall \sigma \cdot \sigma(\text{prop}) = (\text{red}; \sigma)(\text{prop}) \\ \forall \sigma \cdot \sigma \in \text{reachable}(\llbracket C(\text{chaos}(V)) \rrbracket) \\ \implies \sigma(\text{prop}) \end{array}}{\forall \sigma \cdot \sigma \in \text{reachable}(\llbracket C(P) \rrbracket) \implies \sigma(\text{prop})}$$

where $\text{red}(\sigma) \stackrel{\text{def}}{=} (\sigma[\text{grad} := \text{if } (\text{grad} = \text{waitfor}(v)) \text{ then } \text{grad}' \text{ else } \text{grad}]) \not\models \text{pos}$, and $\text{grad}' = (\text{layer}(\sigma) = \text{FIRE}) \text{ then } \text{enabled} \text{ else } \text{delayed}(0)$.

Lemma 3: If $\text{vars}(P) = V$, then $\llbracket P \rrbracket \sqsubseteq_{\text{red}} \llbracket \text{chaos}(V) \rrbracket$.

Lemma 4: VeriSmall programs are monotone with inclusions with variables projections: If $\llbracket P \rrbracket \sqsubseteq_{red} \llbracket Q \rrbracket$ then $\llbracket C(P) \rrbracket \sqsubseteq_{red} \llbracket C(Q) \rrbracket$.

This result follows by checking that all relations used in the semantics commute with *red* as specified in lemma 2, and using structural induction over the program context *C*.

From lemmata 3 and 4, it follows that $\llbracket C(P) \rrbracket \sqsubseteq_{red} \llbracket C(\text{chaos}(V)) \rrbracket$. The desired result then follows from lemma 1.

8 A Small Example

The following example shows how the techniques shown in this paper can be applied to a small example. Consider the following VeriSmall program:

```
initial begin v=0; P1; v=1; wait(w); P2; end;
initial begin w=0; Q1; w=1; wait(v); Q2; end;
```

It should be intuitively clear that if programs P1, P2, Q1 and Q2 do not write to variables *v* and *w*, the programs P1 and Q2 are never executed at the same time. Similarly for P2 and Q1. A proof of this property for Verilog programs is given in [1] using Duration Calculus. However, we can obtain this result more easily by using theorem 1.

Using the semantics given in this paper we obtain a FLTS for the following program:

<pre>initial begin v=0; inP1=1; chaos(a,b,c); inP1=0; v=1; wait(w); inP2=1; chaos(a,b,c); inP2=0; end;</pre>	<pre>initial begin w=0; inQ1=1; chaos(a,b,c); inQ1=0; w=1; wait(v); inQ2=1; chaos(a,b,c); inQ2=0; end;</pre>
--	--

The FLTS is encoded in SMV using a translator we have written (recall that the semantics of a FLTS are independent of the layer information), through which we check that it satisfies the CTL safety property: $AG(\neg(\text{inQ1} \wedge \text{inP2}) \wedge \neg(\text{inP1} \wedge \text{inQ2}))$ — “In every reachable state, *inQ1* and *inP2* are mutually exclusive. Similarly for *inP1* and *inQ2*”². The desired result then follows for programs which use no variables other than *a*, *b* and *c* from theorem 1.

This is not more than a toy example. It is clear that for the actual result we desire, we need a stronger theorem — namely that the programs we replace *chaos(V)* by, may also use variables which are not used in the program context.

² One may object that we have also added the extra assignments in the program, however one can obtain propositions for *inQ1*, *inQ2*, etc in terms of *pos* in the states of the FLTS. Theorem 1 would then also need to be strengthened to allow reasoning about position variables. In our tool, blocks of code can be named so as to automatically produce SMV macros for such properties.

This example only serves to demonstrate how `chaos(V)` can be used to provide more than a simply unconstrained global environment.

Our Verilog-to-SMV translator can handle a much larger subset of Verilog than VeriSmall. In particular it handles non-blocking assignments and edge guards (eg `@(posedge v)`) which allow more interesting examples to be constructed. Some other examples specified and verified include counters, simple arithmetic circuits and small algorithms like the one shown above.

9 Related Work

A number of model checkers come together with an abstract language in which transition systems can be specified. Thus, for example, the SMV [15] input language provides a number of high level mechanisms which can be used to specify transition systems. However, while the language allows means of describing complex transition relations, it is rather limited when it comes to means by which transition systems can be combined together. Similarly, Verus [16] provides a high level language in which transition systems can be specified. However, due to the high level nature of the language, one is then left unsure as to whether the semantics specified match those of a Verilog simulator precisely. Since we also view our language semantics specification as a documentation of the semantics, this is undesirable. Another problem is that the the priority levels inherent in Verilog would have to be encoded within the language, introducing another possible source of errors.

The concept of layers corresponds very closely to the idea of priority in process algebra [17]. Usually, however, priorities are associated to transitions or particular language operators, as opposed to particular states. It would be useful to compare our approach to these alternative ones.

The semantics of Verilog have been expressed in terms of a number of variants of transition systems. It is important to note that Verilog has two different semantic interpretations: simulation semantics (which we deal with) and synthesis semantics (which is used in tools which synthesise Verilog code into hardware). Fiskio-Lasserer *et al* [18] express the simulation semantics in terms of an operational semantics while Sasaki [19] has expressed the semantics in terms of abstract state machines. Both provide an excellent documentation of the semantics of the language but do not seem to be particularly suited for proofs about large programs. Gordon *et al* [20] gives the synthesis semantics of the language in terms of transition systems, and the end result of the interpretation is very similar to the one we present. However, the semantics are expressed in terms of a rather complex compilation process which would be rather difficult to prove that it is semantic preserving with respect to other published semantics. The same problem can be found in [21], where a compilation procedure is given to translate programs into finite state machines.

10 Conclusions

We have presented a set of combinators for enriched transition systems. The most important features of our approach are the compositionality and the abstraction which allowed us to express the semantics of VeriSmall so easily. Also, the full semantics of Verilog are just a scaled up version of the semantics of VeriSmall we give here, which is encouraging when one considers the intricate semantics the language has. This work offers us a myriad of opportunities to explore. One of the priorities is the derivation of a number of laws which allow a guaranteed correct implementation of the combinators used. We have implemented a Verilog-to-transition-system translator based on these semantics, which is available upon request from the author. The translator supports a substantially larger subset of Verilog than the one presented in this paper, including non-blocking assignments and guards.

It is generally accepted that any realistic verification of Verilog or VHDL semantics can only be effectively performed at the synthesis level. It is however the case, that simulation is used extensively, and synthesis semantics are different from the related simulation semantics. We are not advocating the verification of large designs at simulation level, but attempt to provide a framework in which the simulation semantics of languages like VHDL and Verilog can be formally reasoned about.

As can be seen from the main theorem in this paper, certain problem solving techniques seem to recur in different languages. The use of a **chaos** constructor, for example, seems to be applicable to most languages. Furthermore, the proof of correctness of the theorem corresponding to the one we give would, in most cases follow the exact same steps. We hope this also to be the case with other results, especially ones related to the generation of a compiler from the source language to transition systems from a given semantics.

Acknowledgements

Thanks to Koen Claessen and Mary Sheeran for their invaluable comments. Thanks also to the anonymous referees for their helpful comments about the paper and how to improve it. Finally, thanks also must go to Walid Taha without whom the language formalised in this paper would not have been ‘VeriSmall’ but ‘a subset of Verilog’.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS00*, 2000.
2. S. Campos, E. Clarke, and M. Minea. The Verus tool: A quantitative approach to the formal verification of real-time systems. *Lecture Notes in Computer Science*, 1254, 1997.

3. S. Cheng, R. Brayton, R. York, K. Yelick, and A. Saldanha. Compiling Verilog into timed finite state machines. In *1995 IEEE International Verilog Conference (Washington 1995)*, pages 32–39. IEEE Press, 1995.
4. E. M. Clarke. Automatic verification of finite-state concurrent systems. *Lecture Notes in Computer Science*, 815, 1994.
5. R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000. To appear.
6. John Fiskio-Lasseter and Amr Sabry. Putting operational techniques to the test: A syntactic theory for behavioural Verilog. In *The Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, 1999.
7. Mike Gordon, Thomas Kropf, and Dirk Hoffman. Semantics of the intermediate language IL. Technical Report D2.1c, PROSPER, 1999. available from <http://www.cl.cam.ac.uk/~mjc/IL/IL15.ps>.
8. K. McMillan. *Symbolic model checking: An approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
9. Gordon J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In *Proceedings of the Workshop on Formal Techniques in Hardware and Hardware-like Systems, Marstrand, Sweden, June 1998*.
10. H. Sasaki. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. In *Proceedings of DATE'99 (Design, Automation and Test in Europe), ICM Munich, Germany, March 1999*.
11. IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Computer Society Press, Piscataway, USA, 1996.
12. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *CAV00*, 2000.

Sequential Equivalence Checking by Symbolic Simulation

Gerd Ritter

Dept. of Electrical and Computer Engineering
Darmstadt University of Technology, D-64283 Darmstadt, Germany
`ritter@rs.tu-darmstadt.de`

Abstract. An approach for interpreted sequential verification at different levels of abstraction by symbolic simulation is proposed. The equivalence checker has been used in previous work to compare two designs at rt-level. We describe in this paper the automatic verification of gate-level results of a commercial synthesis tool against a behavioral specification at rt-level. The symbolic simulator has to cope with different numbers of control steps since the descriptions are not cycle equivalent. The state explosion problem of previous approaches relying on state traversal is avoided.

The simulator uses a library of different equivalence detection techniques which are surveyed with main emphasis on the new techniques required at gate-level. Cooperation of those techniques and good debugging support are possible by notifying detected relationships at equivalence classes rather than to manipulate symbolic terms.

1 Introduction

Automatic equivalence checking techniques are often limited to verify single or related synthesis steps. Even if the design flow permits a verification after each synthesis step, i.e., a kind of "waterfall" verification, bridging as many synthesis steps as possible during verification increases protection against implementation errors of the verification in practice. The results of several tools each verifying single synthesis steps can be cross-checked. Moreover, a method of step-by-step verification can be problematic concerning modifications of the specification during the design cycle and manual modifications.

The verification of gate-level descriptions is usually done by checking *combinational* equivalence to the corresponding synthesizable description at rt-level. The knowledge about the synthesis tool can be used to facilitate the verification process, e.g., if the independent verification tool "copies" the synthesis steps to obtain a transformed specification which is structurally similar and, therefore, easier to compare with the implementation.

A *sequential* verification over several cycles is required if specification and implementation are not cycle-equivalent. This is often the case when comparing behavioral specifications at rt- or algorithmic-level with gate-level implementations. Different numbers of control steps have to be considered in the implementation to perform a computation or to demonstrate a property of the specification

at a higher abstraction level. But even if a synthesizable structural rt-level design is given, the sequential behavior may deviate, e.g., due to retiming.

Our symbolic simulation approach has been introduced in [24, 25] for equivalence checking of two designs at rt-level which requires in general also a sequential verification, see the dotted line in Fig. 1. This paper describes the improvements of the approach which make a *sequential* verification of a description at *gate-level* against a specification at rt-level possible. The equivalence detection techniques of the symbolic simulator are surveyed putting the main emphasis on the additional techniques required at gate-level. The flexible use of decision diagrams to detect corner cases of equivalences is substantially improved.

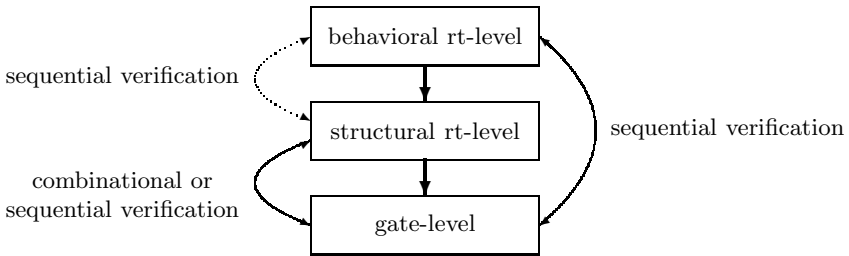


Fig. 1. Verification of synthesis steps

Only small parts of the verification problem are reflected by decision diagrams to avoid graph explosion. An important advantage of the approach is the good debugging support of the tool which can provide meaningful information about a simulation run.

Sequential verification techniques relying on state space exploration cope with different abstraction levels but suffer from the state space explosion problem, which limits their application for equivalence checking. Our symbolic simulation approach permits sequential verification of a design at gate-level against a specification at rt-level avoiding state space traversal. The specification needs to be neither synthesizable nor cycle equivalent to the implementation and the verification process is independent of the synthesis tool used.

Techniques denoted "symbolic simulation" or "symbolic evaluation" have been developed since the 1970s, e.g., in [24, 4]. The following essentials which are explained in more detail in the rest of the paper distinguish our symbolic simulation approach and permit a sequential verification at gate-level:

- detected relationships, e.g., equivalence of terms are notified at equivalence classes; symbolic terms are not rewritten;
- only the information of the equivalence classes of the direct arguments is used in most of the cases to reveal equivalences between terms, i.e., tracing the expression trees of the arguments is avoided to permit a fast simulation;
- simulation is guided along valid, i.e., logical consistent paths in the descriptions instead of reducing the verification problem to a single formula which is checked afterwards;

- several register assignments along a valid path are explicitly distinguished; therefore, term-size explosion is avoided which can occur if only the expressions assigned are considered during simulation;

Our contribution avoids a number of well-known deficiencies of other techniques:

- theorem proving techniques require significant user interaction for our verification problems although they have a larger application area using general algorithms; our verification is automatic;
- techniques depending on state space exploration are not able to cope with the large state spaces of our examples;
- several techniques generate first a single huge formula to be checked afterwards; the formulas resulting from sequential rt-/gate-level verification are too complex for formula checkers even with special support for bit-vector arithmetic; constructing a corresponding decision diagram for the verification problem leads to graph explosion; our techniques use decision diagrams, too, but only to check efficiently small parts of the problem.

Section 1.1 discusses related work. The basic simulation process is briefly reviewed in section 1.2, for more details see [24]. Section 1.3 gives an overview of the equivalence detection techniques used during the symbolic simulation. The more powerful, but less time-efficient equivalence detection based on decision diagrams is described in section 1.4. Experimental results and a conclusion are given in section 1.5 and 1.6.

2 Related Work

The purpose of our symbolic simulator is *automatic interpreted sequential* verification which distinguishes it from approaches performing logic verification or combinational equivalence checking using, e.g., OBDDs or SAT-checker. An interpretation of functions is not necessary for some formal verification problems where specialized approaches can perform better. Note that comparing rt- and gate-level descriptions requires mostly an interpretation.

All techniques which depend on state space exploration face the problem that the number of states grows generally exponentially with the number of storage elements which is known as the *state explosion problem*. This remains a problem even if states and transition relation are represented symbolically by decision diagrams - usually OBDDs - as in symbolic model checking [25], i.e., traversal leads for many designs to graph explosion or long computation times. Various techniques exist to tackle these problems which allow pushing the limit further (e.g., by state reduction [26]) but either do not provide a general solution for fast automatic traversal of large circuits or their area of application is restricted (e.g., [27]).

Techniques generating a single formula for the verification problem which is verified afterwards with a formula checker like SVC [28, 29, 30] have been successfully applied to verification problems described by the dotted line in Fig. 1.

They do not distinguish explicitly the different intermediate symbolic values of the registers: an assignment is considered by using the symbolic term assigned whenever the register is used in the following. This can lead to term-size explosion especially at the structural level. In general, an application to gate-level descriptions is not possible since in each step the whole gate-level expression has to be substituted and the resulting formula can not be checked even with bit-vector arithmetic decision procedures. Furthermore, the information about the sequential behavior gets lost. Therefore, we do not replace in our approach the intermediate register values but distinguish them only by indices, see section 4. SVC is more efficient for "uninterpreted" verification where they usually outperform interpreted techniques.

Symbolic Trajectory Evaluation (STE) [24, 25] is an efficient model checking approach which reasons about Trajectory Formulas, i.e., a restricted temporal logic which combines Boolean expressions and the "next-time" operator. An assertion ($A \Rightarrow C$) is verified by simulating the system over the weakest trajectory for A which is a possible behavior of the model and checking adherence to C . STE operates on *symbolic* ternary values, parameterized in terms of a set of Boolean variables which encode a symbolic value for different operating conditions, e.g., the behavior of an inverter can be specified by [**in is** $a \Rightarrow \text{N(out is } \neg a)$]. Usually two OBDDs are used to represent each symbolic node value.

An advantage of STE over other model checking algorithms is that it is sensitive to the *property* to be verified rather than to the state space. It has been successfully applied to the verification of large memory arrays (e.g., [26, 27, 28]) at the transistor-level. Properties of datapath components like multipliers or systolic arrays [29] and of the IntelTM instruction marker [30] have been verified *with user interaction* using the VossProver which combines STE and theorem proving. The verification of complex industrial floating-point designs at gate-level were performed with Forte, an evolution of Voss, but required significant human effort [31]. A decomposition of the verification task into smaller parts by data-space partitioning is used in [32] to allow for an automatic verification of floating-point units and of an IntelTM instruction marker using Voss.

Although well suited to verify functional properties of *data intensive* parts, an application of STE to the verification of complex *control* systems with data operations against a specification at higher level is not clear, for example, concerning the effectiveness of the representation of the symbolic ternary values by decision diagrams. Furthermore, the restricted logic constrains the applicability to equivalence checking.

Two related heuristics for sequential verification are proposed by [33, 34]. Numerical and symbolic simulation are combined in [35]. In each clock cycle, parts of the inputs are tied automatically to constants (as in numerical simulation) while others get symbolic values. Graph-explosion of the OBDDs is avoided because of the constant inputs while the number of test vectors simulated in one time unit is significantly increased compared to numerical simulation. The objective of [36] is to find efficiently counterexamples to safety properties by using iteratively

¹ [37] gives a good introduction.

numerical simulation, OBDDs and ATPG. The circuit is simulated and nodes which remain unchanged are remarked. A heuristic "solver" using OBDDs and ATPG techniques with a defined computation limit, generates inputs enabling transitions which have not been taken yet. These results are used to guide the numerical simulation in the next step. This second approach especially is related to our symbolic simulation by using a mixture of different techniques alternating. However, the intention of both approaches is different since they sacrifice in their heuristics completeness of the verification process in order to allow rather a fast "falsification" without guaranteeing that corner cases are considered.

3 Simulation Algorithm

Our sequential verification technique compares two *acyclic* descriptions. Loops have to be duplicated previously according to the *maximum* number of executions. For many cyclic designs with infinite loops the verification problem can also be reduced to the equivalence check of acyclic sequences, see [24].

The symbolic simulator is used to check the *computational equivalence* of two designs. Two descriptions are computationally equivalent if both produce the same final values on the same initial values. For instance, the two descriptions in Fig. 2 are computationally equivalent, if `ctrl` is initialized with 0 and if the execution takes two cycles. The implementation at gate-level comprises the signal assignments to the three bits of the register `r` and to the control flag `ctrl`. Two cycles of symbolic simulation are required to demonstrate equivalence. In

Specification	Implementation
<code>r ← r+1;</code>	<code>r[2] ← not(ctrl and m) and (r[2] xor (r[1] and r[0]))</code>
<code>if m=0</code>	<code>r[1] ← not(ctrl and m) and (r[1] xor r[0])</code>
<code>then r ← r+1;</code>	<code>r[0] ← not(ctrl and m) and (not r[0])</code>
<code>else r ← "000";</code>	<code>ctrl ← not(ctrl)</code>

Fig. 2. Example of two computationally equivalent descriptions

the first cycle, `r+1` is calculated and `ctrl` is set true. The *if-then-else* evaluating the flag `m` is considered in the next cycle. Symbolic simulation has to demonstrate that the final values of `r` are the same.

A brief overview of the basic simulation algorithm presented in more detail in [24] is given in the following. The modifications necessary for verification at gate-level are described.

The different values of the registers after assignments are distinguished in our approach by indexing rather than to substitute the register in the following by the symbolic term assigned which can lead to term-size explosion, see section 4. An indexed register name is called a *RegVal*. A new *RegVal* with an incremented index is introduced after each assignment to a register. An additional

² An empty loop body is simulated if the number of executions is smaller.

upper index s or i distinguishes the *RegVals* of specification and implementation. For example, $\mathbf{ar} \leftarrow \mathbf{a} + \mathbf{b}$; is replaced by $\mathbf{ar}_2^s \leftarrow \mathbf{a}_1^s + \mathbf{b}_1^s$; in the specification if all registers have been already assigned once. Only the initial *RegVals* as anchors are identical in specification and implementation, since the equivalence of the two descriptions is tested with regard to arbitrary but identical initial register values.

Two cycles have to be simulated symbolically in the example of Fig. 1. Therefore, the gate-level description representing only one cycle is put together two times before indexing the registers, i.e., the description is duplicated accordingly to the number of cycles required.

Each symbolically executed assignment establishes an equivalence between the destination *RegVal* on the left and the term on the right side. Additional equivalences between terms are detected during simulation using the techniques presented in section 1 and 2. Two terms or *RegVals* are equivalent \equiv_{term} if under the decisions C_0, \dots, C_n taken previously on the path, their values are identical for all initial *RegVals*. Case-splits lead to the decisions C_0, \dots, C_n , which constrain the set of possible initial *RegVals*. Equivalent terms are detected along valid paths, and collected in equivalence classes (*EqvClasses*). We write $term_1 \equiv_{sim} term_2$ if two terms are in the same equivalence class established during simulation. If $term_1 \equiv_{sim} term_2$ then $term_1 \equiv_{term} term_2$. Initially, each *RegVal* and each term gets its own equivalence class. Equivalence classes are unified if two terms are identified to be equivalent by reasoning, after assignments, or in case splits. Equivalence classes permit to keep also track about *unequivalences* of terms. Two terms or *RegVals* are unequivalent $\not\equiv_{term}$, if their values are never identical for all possible *RegVals* under the decisions C_0, \dots, C_n taken preliminary on the path. We write $term_1 \not\equiv_{sim} term_2$ if two terms are identified to be $\not\equiv_{term}$ during simulation or set unequivalent in a case-split. Equivalence classes containing $\not\equiv_{sim}$ terms are unequivalent. Note that *EqvClasses* containing distinct constants are unequivalent since a constant is represented by only one *EqvClass*.

Fig. 1 gives a simplified overview of the symbolic simulation algorithm which has been implemented iteratively for optimization, see [10] for more details. Specification and implementation are simulated in parallel. A case-split is performed when simulation reaches a condition C that cannot be decided in general but depends on the initial register values (lines 1 and 2). The information of the *EqvClasses* is used to decide conditions at branches consistently, i.e., to avoid unnecessary case-splits which lead to false paths. Note that *equivalence_check* is called recursively in line 3 with only those parts of *spec* and *impl* which are not simulated yet.

A complete path is found when the end of both descriptions is reached. The computational equivalence of the descriptions in this path is tested by checking whether the relevant final *RegVals* are in the same *EqvClass* (line 4). This test may fail since the equivalence detection during the path search is not complete to permit a fast symbolic simulation. Therefore, more accurate tests called *dd-checks* based on decision diagrams [11] are used at the end of a path (line 5).

equivalence_check(spec, impl)

1. $\left\{ \begin{array}{l} \text{Simulate } spec \text{ and } impl \text{ in parallel and} \\ \text{perform intermediate } dd\text{-checks if necessary} \end{array} \right\} \text{ until}$
 - (a) a condition C is reached that cannot be decided in general but depends on the initial register and memory values, or
 - (b) the end of both descriptions is reached.
2. **if** a condition C blocks **then**
3. **RETURN** $(equivalence_check(spec, impl) \mid_{C=FALSE}) \wedge (equivalence_check(spec, impl) \mid_{C=TRUE})$
4. **elseif** final values of registers are equivalent **then** **RETURN**(TRUE)
5. **else** perform *dd-checks*;
6. **if** (final values of registers are equivalent) \vee **then** **RETURN**(TRUE)
7. (a condition has been decided inconsistently)
8. **else** **RETURN**(FALSE)

Fig. 3. Simplified algorithm of the symbolic simulation

They have to reveal whether computational equivalence is given in this path but was not detected (line 4), a condition has been decided inconsistently due to the incomplete equivalence detection on the fly (line 6), or a valid counter-example is found (line 8). All relevant information of the path can be resumed in the latter case to facilitate debugging.

Intermediate *dd-checks* are often useful (line 6) if one of the descriptions is at gate-level rather than if both descriptions are at algorithmic- or rt-level. The same entire Boolean expressions assigned to the register bits have to be simulated at gate-level in each symbolic simulation cycle. It is crucial to find relationships of the *preceding* values of the control registers in order to detect equivalences in the *next* cycle between the Boolean expressions at gate-level and the much simpler corresponding terms in the specification at algorithmic- or rt-level. The *final dd-checks* in line 8 become impractical if the "link" between specification and implementation gets lost early on the path: too many intermediate simulation cycles at gate-level have to be considered in the decision diagrams before equivalent terms of specification and implementation are reached, see also section 4. Therefore, the *dd-checks* are also used during the path search if no equivalence has been found yet for a term assigned to a *RegVal*. It is useful if the user restricts the application of those intermediate tests by simply denoting the control registers. Note that the verification process is automatic and requires no insight of the user.

4 Equivalence Detection Techniques

4.1 Preliminaries

Our equivalence detection during simulation or on the fly is not complete since it would be too time-consuming to check all possible equivalences of terms. On

the other hand, it should be sufficiently powerful so that in most of the cases the more accurate, but slower *dd-checks* described in section 4.1 are not required. These should only reveal special cases of equivalence which occur seldom or are hard to detect. Note that one reason for the inferior speed of the decision diagram based *dd-checks* is that a limited backtracking of the simulation is required. All other techniques use in general just the current state of the *EqvClasses* of the direct arguments to detect equivalences between terms, i.e., they avoid a time-consuming backtracking of the expression trees.

A basic concept of our symbolic simulator is that terms are never rewritten or exchanged; only the information of the corresponding *EqvClass* is modified or *EqvClasses* are unified. Therefore, no unique representation is required which easily allows to add new equivalence detection techniques and which permits a hierarchical equivalence detection according to the principle of Hennessy and Patterson [10]: "Make the common case fast".

4.2 General Equivalence Detection

Mostly, specific equivalence detection methods developed for a function are faster and more accurate than general approaches. However, general techniques have to be provided since no function-specific rule may apply or no specific technique exists, e.g., for user-defined functions.

It holds for all functions, that two terms are equivalent, if the function symbol is the same and all arguments are pairwise equivalent, i.e., the *EqvClasses* of the arguments are pairwise identical. A weaker condition can be used, if the function is symmetric.³ Testing if any argument has an equivalent counterpart *in one direction* is not sufficient since the number of arguments can vary, e.g., $x+1+1 \not\equiv_{sim} x+1$ and the same argument can be used twice, e.g., $x+1+1 \not\equiv_{sim} x+1+2$. Therefore, the *occurrences* of the *EqvClasses* of the arguments have to be the same on both sides. The checks are simplified, if the number of arguments is fixed.

The set of *candidates* for testing equivalence is determined by two approaches. For user-defined or not frequently used functions, all terms with the same function symbol found during simulation on a path are collected. They represent the candidates for checking equivalence of a new term with the same function symbol. Note that this set consists in general only of a small fraction of all terms with this function symbol since it is *path-dependent*. However, this approach is inefficient for frequently used functions, especially concatenation and single-bit-selection at gate-level. A smaller set of candidates is determined for those functions by examining the *EqvClasses* of the arguments. Consider first a function with a single argument. Two terms are equivalent if the *arguments* are in the same *EqvClass*. Therefore, candidates can be determined by evaluating the *EqvClass* of the argument of a new term, i.e., candidates must

1. have an argument which is a member of this *EqvClass*,

³ The techniques described in this section are not described as "uninterpreted" because the symmetric property is evaluated.

2. use the same function symbol, and
3. have been found on the actual path.

Each of these terms is equivalent to the new term for functions with only one argument. Otherwise property [14](#) must hold for every argument, considering whether the function is symmetric or not. The set of candidates can be determined easily since the information about which functions use a term as argument is marked at the term during pre-processing separately for each function symbol. Note that equivalence of the arguments need not be a necessary condition if other function-specific properties among symmetry are considered.

The second approach is less efficient if the function-specific equivalence detection techniques are mostly successful, e.g., for Boolean- or arithmetic-functions, or if only few terms of the same function are encountered *on the same path*. Therefore, it is mainly used for concatenation- and bit-selection. It would become slow, if the *EqvClass* of an argument has many members which is common for the *EqvClasses* of the constants 0 or 1. But these corner-cases are mostly considered separately by the specialized equivalence detection techniques.

4.3 Bit-Vector Functions

Bit-vector functions take bit-vectors as arguments and return a bit-vector or one bit as a result. Common examples are:

- arithmetic functions, e.g., addition, multiplication or subtraction. The equivalence detection for addition is described as a representative for bit-vector functions in more detail below;
- memory operations, i.e., **store**- and **read**-operations; our equivalence detection copes with distinct order of memory operations which is described in detail and compared to other approaches in [22](#);
- bit-selections (e.g., **ir**[7:4]) are considered as operations which return a new term. The result is constant if the term of the selection is constant or the sliced part is overlapped by a constant region. Otherwise eventually partial constant regions are determined and the techniques described in section [15](#) are applied. The constant indexes are compared directly since all indirect selections are considered as memory operations. Note that the index of one-bit selections is considered in the function symbol, see section [15](#);
- concatenation of bit-vectors which occurs often at gate-level since the corresponding register assignment is obtained during pre-processing by concatenating the respective (in general complex) Boolean expressions bit_i , i.e., $\mathbf{reg} \leftarrow (bit_n \& (\dots \& (bit_2 \& (bit_1 \& bit_0)) \dots))$. The parentheses consider the recursion scheme since the concatenation takes only two arguments. For example, first $bit_1 \& bit_0 \equiv_{sim} \mathbf{pc}[1:0]$ is detected during simulation if the expression assigned to **reg** is equivalent to **pc**, then $bit_2 \& (bit_1 \& bit_0) \equiv_{sim} \mathbf{pc}[2:0]$ and so on. Note that the bit-selections may not appear explicitly as terms

⁴ Single-bit selections have only one argument since the constant index is considered in the function symbol.

in the descriptions. A more detailed example is given in section 10.1. Some bit-vector-operations are transformed during pre-processing using concatenation, e.g., a left-shift shifting in 1 becomes $\text{lsh}(a, 1) \rightarrow a[31 : 1] \& 1$;

- uninterpreted functions, e.g., user-defined functions for which only the techniques presented in section 10.1 apply. A special case are **unknown-value**-terms which are guaranteed to be neither \equiv_{sim} nor \neq_{sim} to another term and permit the user to leave, e.g., implementation dependent parts of the design unspecified or unconsidered;
- multiplexers are interpreted as functions with N control bits which select one of 2^N data-words. A transformation into an adequate *if-then-else*-structure is feasible, but blows up the descriptions and can lead to term-size explosion in other approaches, if the overall formula is built in advance and verified afterwards. The *EqvClass* of a **multiplexer**-term and of the selected data-word can be unified if every control bit is equivalent to a constant. The latter is guaranteed by adding a *single* special *if-then-else*-structure in front of each **multiplexer**-term to force a decision about the values of the control bits.
- Boolean operations applied on bit-vectors; only a part of the simplification techniques presented in section 10.1 can be applied on bit-vectors;
- comparison functions, i.e., $>$, $<$, $>=$, and $<=$, can be decided if the arguments are constant or if they are equivalent. Otherwise the information about the range of the argument, marked as *valuebounds* at the *EqvClasses*, is evaluated. The range of terms is mainly restricted by deciding conditions, e.g., $a < 30$ but also for example by arithmetic operations or concatenation, e.g., the four-bit vector $00 \& a_1 \& a_0$ is guaranteed to be less than 4. Two terms are compared by examining pairwise the *valuebounds* of the corresponding *EqvClasses*, which can be incompatible, compatible or indifferent concerning the actual comparison operator. The equivalence detection is possibly used recursively, e.g., the comparison $a_2^s < b_2^s$ assuming the *valuebounds* $a_2^s < x_1^s$ and $b_2^s > y_1^s$ is satisfied if $x_1^s \leq y_1^s$ holds. Comparisons between two arithmetic operations are considered, but often permit no decision if they are modulo, e.g., $(a + 4 < a + 3)$ may hold due to an overflow.

Example for Bit-Vector Functions: Addition

Many arithmetic functions used in *hardware*-designs are modulo, either explicitly or implicitly, e.g., storing the result of an addition in a register without carry. Equivalence detection for the addition with carry-input but *without* carry-output $\text{ADCMOD}(a, b, \text{carry})$ is described in more detail as an example for bit-vector functions.

If all the arguments of an ADCMOD -term are constant then the constant result of the term is calculated and the corresponding *EqvClasses* are unified. Note that this can make the dynamic creation of an *EqvClass* necessary since *EqvClasses* are built during pre-processing only for constants appearing explicitly in the descriptions.

If the **carry** of the term is equivalent to 0, i.e., it is irrelevant then the equivalence detection for symmetric functions for the addition *without* input carry (ADDMOD) is called. If moreover one of the summands is equivalent to 1 then the

result is the same as incrementing the remaining non-constant argument, i.e., any **INCMOD**-term with an \equiv_{sim} argument is equivalent. The same holds if the carry is equivalent to 1 and one of the summands is equivalent to 0. Note that although the equivalence detection is reduced in those cases to check equivalence for **INCMOD** respectively **ADDMOD**, equivalence to another **ADCMOD**-term will be still detected since its arguments would satisfy the same properties.

If the carry and one of the summands is equivalent to 0 then the *EqvClasses* of the **ADCMOD**-term and the remaining non-constant argument are unified. Otherwise the general equivalence detection technique described in section 4.3 is used considering the carry.

Equivalence of successive additions is considered by accumulating the constants and collecting the non-constant arguments. For example, if $x_1^i \leftarrow a+b+4$ holds then $5+x_1^i$ has the accumulated constant 9 and the *positive* non-constant part $\{a,b\}$. Two terms are equivalent if the non-constant parts are equivalent and the accumulated constants are equal. An extension of this concept to include subtraction etc. must carefully consider over- and underflows, which limits the application substantially.

4.4 Boolean Functions

Detecting equivalences of Boolean functions is especially important at gate-level. In the following, **and**-terms are taken as an example. As for all other functions, first properties which are fast to identify and which often occur are checked. For Boolean functions, first constant bits are determined. For example, equivalence of an **and**-term is obvious if one argument is equivalent to 0 or only one argument is not equivalent to 1. Searching for constants is not sufficient in the example of Fig. 4 where $(ak[0] \text{ nand } 1) \text{ and } (\text{not}(ak[0]) \text{ nor } 0) \equiv_{sim} 0$ has to be detected to reveal the constant value of res_1^i . The simplifications during pre-processing

Specification	Implementation
if $x = "0110"$ then $b_1^s \leftarrow ak$; else ... if ... then ... else $c_1^s \leftarrow ak[3:0]$;	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> <i>previously detected:</i> $b_1^i \equiv_{sim} b_1^s, c_1^i \equiv_{sim} c_1^s$ </div> $res_1^i \leftarrow (b_1^i[0] \text{ nand } x[1]) \text{ and } ((\text{not } c_1^i[0]) \text{ nor } x[0]);$

Fig. 4. Example for equivalence detection for boolean functions

can not consider this relationship since it is path-dependent and a result of the previous sequential assignments. Constructing and evaluating the expression $(x \text{ nand } 1) \text{ and } ((\text{not } x) \text{ nor } 0)$ is feasible but violates the recursive scheme of equivalence detection: just the information of the *EqvClasses* of the *direct* arguments is evaluated in order to avoid a slowdown of the simulation if the depth of the Boolean expressions is greater than in this simple example.

The difficulty of the example is that some of the subterms are not constant. But those subterms are either equivalent to $ak[0]$ or to $(\text{not } ak[0])$, i.e., this means that they are *positive- or negative-bit-equivalent* to $ak[0]$ which can easily be remarked at the *EqvClasses* during symbolic simulation:

- $b_1^i[0]$ is equivalent to $ak[0]$ and $x[1] \equiv_{sim} 1$; therefore, the **nand**-term is identified to be *negative-bit-equivalent* to $ak[0]$;
- the term $c_1^i[0]$ is *positive-bit-equivalent* to $ak[0]$; that is why $(\text{not } c_1^i[0])$ is *negative-bit-equivalent* and $((\text{not } c_1^i[0]) \text{ nor } x[0])$ *positive-bit-equivalent* to $ak[0]$; note that the *EqvClasses* of the **nand**-term and of $(\text{not } c_1^i[0])$ are unified;
- finally, the arguments of the **and**-term are *positive*- and *negative-bit-equivalent* to the same bit and, therefore, both can never be satisfied.

Remarking *positive*- or *negative-bit-equivalence* at the *EqvClasses* is not only used to detect contradictions. For example, if an argument is *negative-bit-equivalent* to a bit and all other arguments are equivalent to 1, then the **and**-term is equivalent to the negation of this bit. More applications to **and**-terms and other Boolean functions are straightforward. Finally, the general equivalence detection for symmetric functions is called only with the non-constant arguments (all other arguments are \equiv_{sim} to 1) if equivalence has not yet been detected.

Positive- or *negative-bit-equivalence* has to be propagated even if the term is equivalent to a constant in order to detect equivalences of concatenations, an example is given below. A heuristic is used if the arguments are *positive*- or *negative-bit-equivalent* to different bits. If all but one argument are constant or eliminate each other because they are *positive*- and *negative-bit-equivalent* to the same bit then the information of the remaining argument is propagated. Otherwise the *positive*- or *negative-bit-equivalence* to be propagated is selected according to the following priorities:

1. the bit is equivalent to a bit of an initial *RegVal*;
2. the bit is not in an *EqvClass* with a constant;
3. if two bits are bit-selections from two terms, then the one where not all bits of the selected term are constant is preferred.

Criterion 2. and 3. consider that the register assignments are concatenations of complex Boolean expressions at gate-level. Correct propagation is crucial to identify equivalence to simpler terms of the specification at the top level, Fig. ■ gives an example where equivalence of res_1^s and res_1^i has to be detected. The

Specification	Implementation
<pre> if (ir[0]='1' and mad="0101") then res₁^s ← ir; else ... </pre>	<pre> res₁ⁱ ← ((...) or (not (mad[3]) and ir[3])) & ((...) or (mad[2] and ir[2])) & ((...) or (not (mad[1]) and ir[1])) & ((...) or (mad[0] and ir[0])) & </pre>

Fig. 5. Priority example for propagating *positive*- or *negative-bit-equivalence*

hidden terms in parathenses (...) on the implementation side resume the assignments on other paths of the specification and are assumed to be equivalent to 0 on the actual path. *Positive*- or *negative-bit-equivalence* to $ir[3]$, $ir[2]$, and $ir[1]$ respectively are propagated for the most significant bits of res_1^i since the

other arguments of the **and**-terms (**mad**[3],**mad**[2],**mad**[1]) are constants. But *both* arguments **mad**[0] and **ir**[0] of the least significant bit are equivalent to 1. However, it makes more sense to propagate **ir**[0] following criterion 3: all bits of term **mad** are constant, i.e., equivalence to **mad** could be detected after concatenating without the knowledge of *positive*- or *negative-bit-equivalence* since all other bits are equivalent to constants, too. Therefore, **ir**[0] is propagated and equivalence to **ir** is detected after concatenation.

Standard cells, e.g., the A02-cell of the Alcatel™ MTC45000-library are currently broken during pre-processing using basic Boolean functions, i.e., (A **and** B) **nor** (C **and** D). Simulation speed can be optimized by providing specialized equivalence detection routines also for those standard cells.

4.5 Unequivalences Forcing Terms to Be Constant

Unequivalences can force a term to be constant. Since the domain of a n -bit-vector is restricted to 2^n values, setting it $\not\equiv_{sim}$ to $2^n - 1$ values implies equivalence to the remaining value. Fig. 4(a) gives an example for a bit-vector, where $b \not\equiv_{sim} 10$ and $b \not\equiv_{sim} 00$ and $b \not\equiv_{sim} 11 \Rightarrow b \equiv_{sim} 01$ holds. Note that there can be intervening assignments and other conditions in Fig. 4. Two *EquivClasses* are unequivalent either because of a decision in a case split or since they contain different constants which is not relevant here. Checking after each decision whether the *EquivClass* is set unequivalent to 2^{n-1} constants is not sufficient since also decisions about *parts* of a term have to be considered, see Fig. 4(b) where $a[3] \equiv_{sim} 1$ and $a[2:1] \not\equiv_{sim} 11$ and $a[2:1] \not\equiv_{sim} 10 \Rightarrow a[3:2] \equiv_{sim} 10$ has to be detected. Two counters *ctrl-zero-bit* and *ctrl-one-bit* are introduced for each bit of a term

<p>(a) if b="10" then ... elsif b="00" then ... elsif b="11" then ... else b="01" <i>is true</i></p>	<p>(b) if a[3] then if a[2:1]="11" then ... elsif a[2:1]="10" then ... else a[3:2]="10" <i>is true</i></p>
---	---

Fig. 6. Terms being constant due to decided unequivalences

appearing in conditions and initialized during pre-processing with 2^{N-1} where N is the length of the term (not $2^N - 1$!). A bit i of a term is equivalent to 0 (1) if *ctrl-zero-bit_i* (*ctrl-one-bit_i*) is zero. The counters are decremented if:

- the term is set unequivalent to a constant. *ctrl-one-bit* and *ctrl-zero-bit* are decremented at all bit-positions, where this constant is 0 or 1, respectively;
- a bit-selection of the term is set unequivalent to a constant. Not only the *ctrl-one-bit*- and *ctrl-zero-bit*-counters of the term representing the bit-selection, e.g., **a**[2:1] are decremented but also the corresponding counters of the entire term **a** are set down according to the size of the bit-selection. Multiple selections, e.g., (**a**[10:2])[2:1] are considered by recursion.

Every time a new constant bit is found it is checked whether the whole term is constant, too.

5 Tests Using Decision Diagrams

5.1 Checking Formulas by Means of Decision Diagrams

Decision diagrams are used in the *dd-checks* to reveal equivalences which are not detected by the techniques presented in the previous sections, examples are given in section [2.1](#), [2.2](#) and [3.1](#). Two tests are provided:

- testing whether two terms are equivalent; note that checking the validity of a condition is the same as comparing it to the constant 1;
- testing whether a term is equivalent to a constant; this is a different case since the value of the constant is unknown.

The formulas built are verified using *vectors* of OBDDs [\[10\]](#) where each graph represents one bit of the two terms to compare. The competitiveness of vectors of OBDDs using the TUDD-package [\[11, 12\]](#) has been demonstrated even for large bit-vectors compared to *BMDs and the automatic formula checker SVC in [\[13\]](#). The main advantage compared to *BMDs is that bit-selection is almost "for free" while this operation is worst-case exponentially using *BMDs. Bit-selection is used frequently in practical examples, either explicitly or implicitly. Vectors of OBDDs can compete with automatic formula checkers like SVC on bit-vector arithmetic problems since specific paths are chosen, i.e., the control flow is mainly determined. Furthermore, OBDD-vectors have the same efficiency on gate-level verification, where automatic formula checkers are limited: case-explosion is possible if case-splits are necessary on the single bits of a bit-vector and term-size explosion results from building the formula over several sequential steps in advance and verifying it afterwards. Note that a verification using *only* vectors of OBDDs without considering results of the symbolic simulation is neither efficient nor feasible for large examples, see section [4.1](#).

5.2 Setting Cutpoints for *dd-checks*

The variables occurring on both sides have to be the same if the equivalence of two terms is tested using the decision diagrams. This can be achieved by backward-substitution so that only initial *RegVals*, which are identical in specification and implementation, or constants occur on each side. Note that the formula is less complex than a formula describing the entire verification problem since a specific path is chosen. However, a complete backward-substitution is not efficient since only the information about the path but not about equivalences detected by the other techniques is used. For example, if both terms depend only on two intermediate *RegVals* detected previously to be equivalent, it makes sense to introduce a *dd-cutpoint* and to consider this *dd-cutpoint* as primary input: all expressions or assignments previous to this *dd-cutpoint* do not have to be considered in the decision diagrams.

Therefore, first the two sets representing all *EqvClasses* of the intermediate terms are collected in a fast backtracking. The intersection of those two sets of

EqvClasses represents the candidates for *dd-cutpoints*. Any term with an *EqvClass* in the intersection is represented by a *dd-cutpoint* when constructing the formula by backward-substitution, i.e., the *dd-check* consider this term as a primary input just as the initial *RegVals*.

The *dd-cutpoints* have to be removed in some cases since they hide subterms which are required to demonstrate equivalence, see for an example section [4.1.1](#).

5.3 Intermediate *dd-checks*

Section 4 motivated the use of intermediate *dd-checks* at gate-level *during* the path search (line 11 in Fig. 4) instead of using them only at the end of a path. Fig. 4 gives an example. The register `cnt` is assumed to be a microprogram counter

Specification	Implementation
if $ak[3:0]=mi[3:0]$	$cnt_1^i \leftarrow bit_n \ \& \dots \&$
then ...	$((ak[3] \text{ xor } mi[3]) \text{ nor } (ak[2] \text{ xor } mi[2]))$ and
else <i>selected branch</i> ;	$((ak[1] \text{ xor } mi[1]) \text{ nor } (ak[0] \text{ xor } mi[0]))$
	$\& \dots \& bit_0;$

Fig. 7. Example for the advantages of intermediate *dd-checks*

and the assignments to all registers depend on the value of this control register. The assignment to `cnt` is represented at gate-level by a concatenation (&) of the single bits. Only the expression of one bit is shown in Fig. 1. This bit is constant since $\text{ak}[3:0] \neq_{sim} \text{mi}[3:0] \Rightarrow ((\text{ak}[3] \text{ xor } \text{mi}[3]) \text{ nor } \dots (\text{ak}[0] \text{ xor } \text{mi}[0])) \equiv_{sim} 0$ which is not revealed without *dd-check* by the other equivalence detection techniques.

Assume that the assignments to all registers in the next cycle can be identified to be equivalent to a corresponding *RegVal* in the specification without *dd-check* only if the (controlling) microprogram counter is detected to be equivalent to a constant. Otherwise the "link" between terms in specification and implementation gets lost not only in the next cycle but also in all succeeding cycles since the respective preceding *RegVals* are used as arguments. A final *dd-check* would be complex since all cycles have to be considered.

Loosing the "link" is avoided by providing an intermediate *dd-check* at gate-level if no equivalence between the term assigned to a *RegVal* and any other term has been detected by the other equivalence detection techniques. This intermediate *dd-check* reveals in the example of Fig. 4 that cnt_1^i is constant. The user can limit the application of intermediate *dd-checks* on relevant control registers. This (easily provided) information is optional, but can decrease simulation time significantly.

The *dd-check* requires an assumption about which term might be equivalent to the intermediate $RegVal^i_x$. If the register does not exist in the specification, e.g., a control register of the hardware implementation, it is only checked

- whether the term is equivalent to a constant. The *dd-vector* of the term is built using each *RegVal* of the previous cycle as *dd-cutpoint*. If each bit of

the *dd-vector* is equivalent either to 0 or 1, then the constant result of the term is calculated;

- if the term has not changed in the last step, i.e., $RegVal_x^i \equiv_{sim} RegVal_{x-1}^i$

Otherwise equivalence to the first corresponding $RegVal_y^s$ in the specification (with the lowest number y) is checked, which is neither equivalent to some term of the implementation nor to some initial *RegVal*. Consider first the case that the preceding $RegVal_{x-1}^i$ in the implementation has an equivalent "counterpart" in the specification. In this case, all *RegVals* of the preceding cycle are used as *dd-cutpoints* in the implementation during the *dd-check*. But equivalent terms might be reached in specification and implementation after a different number of cycles. For example, $x_1^s \leftarrow a+b+c$ in the specification is calculated in two cycles by $x_1^i \leftarrow a+b$ and $x_2^i \leftarrow x_1^i+c$ in the implementation. Only x_2^i has an equivalent counterpart in this case. The *dd-check* cannot reveal this fact if the *dd-cutpoints* are set to the previous cycle, i.e., x_1^i . Therefore, a failed *dd-check* is repeated with the *dd-cutpoints* shifted successively to the preceding cycle until either the *dd-check* is satisfied or the *RegVal* of the relevant register in the implementation has an equivalent counterpart in the specification. Note that equivalence would be revealed in this simple example used for demonstration without *dd-check* by the techniques described in section 4.

5.4 Considering Previous Decisions

A case-split is performed each time the value of a condition depends on the initial values of the registers. The decision is reflected in the *EqvClasses* and is, therefore, considered by the equivalence detection techniques during the symbolic simulation as well as during the construction of the formula in a *dd-check*. There remain cases where the decisions have to be provided separately, Fig. 8 gives a simple example. The equivalence of the final values of **res** is not detected

Specification

```
if m=0110 or m=0011 then ...
elseif m=0010 or m=0111 then ...
else res_1^s ← b[31:1]&0;
```

Implementation

```
res_1^i ← b[31:1]&((not m[3]) and m[1]);
```

$$(m \not\equiv_{sim} 0110) \text{ and } (m \not\equiv_{sim} 0011) \text{ and } (m \not\equiv_{sim} 0010) \text{ and } (m \not\equiv_{sim} 0111) \\ \Rightarrow ((\text{not } m[3]) \text{ and } m[1]) \equiv_{sim} 0$$

Note that none of the bits of **m** is constant.

Fig. 8. Considering decisions in the *dd-check*

without *dd-check* since none of the bits of the bit-vector **m** is constant. But the *dd-check* has to consider the unequivalences of **m** and the four constants to reveal that the least significant bit of **res** is equivalent to 0 (see box in Fig. 8).

Therefore, every *dd-check*, which failed to demonstrate an equation called *equ_to_check*, is repeated considering decisions about conditions which share terms with the formula, i.e., $decisions \Rightarrow equ_to_check$ is checked. Note that

only previous decisions are considered for intermediate *dd-checks*. If it is only checked whether a term is equivalent to a constant, the test has to be refined by checking if *decisions* implies that each bit of the term is either 0 or 1:

$$\forall i \in \text{bits of term} : [\text{decisions} \Rightarrow \text{dd_of}(i)] \text{ or } [\text{decisions} \Rightarrow \text{not}(\text{dd_of}(i))]$$

Note that accessing *dd_of(i)* is for free using vectors of OBDDs.

Terms are often represented by *dd-cutpoints* or by other (simpler) terms in the same *EqvClass* in the formulas constructed for the *dd-check*. For example, if a term is in an *EqvClass* with a constant, then only the OBDD for the latter is constructed. However, these replacements have to be considered when including previous decisions. Assume an additional assignment $\text{ir}_1^s \leftarrow \text{ak}[1:0]$ and a condition **if** $\text{ir}_1^s = "10"$ **then** ... in the example of Fig. 4. The constants 1 and 0 are simpler to represent than the equivalent terms $\text{ak}[1]$ and $\text{ak}[0]$. Therefore, the constants replace those terms in the *dd-check*. But the formula $\text{ak}[3:0] \not\equiv_{\text{sim}} \text{mi}[3:0] \Rightarrow ((\text{ak}[3] \text{ xor } \text{mi}[3]) \text{ nor } (\text{ak}[2] \text{ xor } \text{mi}[2])) \text{ and } ((1 \text{ xor } \text{mi}[2]) \text{ nor } (0 \text{ xor } \text{mi}[0])) \equiv_{\text{sim}} 0$ is not valid. Note that ir_1^s does not appear in the formula. The correct result is obtained if the substitutions are considered during the construction of *decisions*.

6 Experimental Results

Symbolic simulation can be applied to verify the computational equivalence of descriptions at different levels of abstraction:

- *rt-level against rt-level*: the descriptions at rt-level can have different implementational details and the number of steps to calculate a result may vary.
 - *behavioral-rtl against behavioral-rtl*: experimental results for the verification of automatically constructed pipelined processors are presented in [14]. The results in [14] demonstrate that our symbolic simulation copes with distinct orders of memory operations in the two descriptions to be compared.
 - *behavioral-rtl against structural-rtl*: the structural implementation of an architecture with microprogram control has been compared to behavioral specifications in [14]. The implementation details of the structural description and the fact that a different number of sequential steps has to be considered makes verification complex. Although the verification required *dd-checks*, it was not necessary to use them in intermediate steps as described in section 4. Note that also verification results for structural descriptions with different implementational details of pipelined DLX-processors are reported in [14].
- *rt-level against gate-level*: symbolic simulation copes not only with logic verification of cycle-equivalent descriptions but can also be used if computational equivalence of the descriptions is given only after several number of control steps (see below).

- *algorithmic-level against rt- or algorithmic-level* is a current research topic. Programs in a subset of C are translated into our experimental language LLS [10]. Verification is limited by loops which have to be unrolled.

Two types of examples have been examined, a simple read/write-architecture (RWA), which takes three cycles to execute an instruction and a more complex architecture with microprogram control (MPA). Two specifications of the second architecture without cycle-equivalence are given; only the first is used for synthesis; therefore, it is cycle-equivalent to the synthesis result. The gate-level descriptions are generated using the Synopsys® Design Compiler™ with the Alcatel™ MTC45000-library. All memory-operations are replaced by assignments to interfaces before synthesis. Equivalence of memory operations on these ports has been verified according to [10], too. The MPA synthesis result comprises 927 standard cells, two arithmetic units and one incrementer. The standard cells except the arithmetic blocks and the memory are broken internally into basic Boolean functions with up to 4 inputs. Tab. 1 summarizes the results. All our measurements are on a Sun Ultra II with 300 MHz. Four equivalence checks have been performed:

- (1) 1 cycle RWA^{RTL} against 1 cycle RWA^{gate}
- (2) one instruction (3 cycles) RWA^{RTL} against one instruction (3 cycles) RWA^{gate}
- (3) 1 cycle synthesizable specification MPA_{cycle}^{RTL} against one cycle MPA^{gate}
- (4) 1 instruction with $m \leq 8$ cycles in the non-synthesizable specification without cycle-equivalence $MPA_{non-cycle}^{RTL}$ against 1 instruction with $n \leq 10$ cycles in MPA^{gate} ; m and n depend on the instructions and may be different.

The verification time given in Tab. 1 increases for both designs acceptably with the number of sequential steps simulated. Especially the last check would lead to term-size explosion if the formula is built in advance and evaluated afterwards, since the whole gate-level expressions of a cycle represent the arguments in the next cycle. The number of required *dd-checks* is given in the fifth column of Tab.

Table 1. Experimental results

check number	cycles		Verification time	<i>dd-checks</i>
	spec	impl		
(1) RWA (one cycle)	1	1	1.7s	-
(2) RWA (one instruction)	3	3	5.5s	-
(3) MPA (with cycle-equiv.)	1	1	74 s	13
(4) MPA (w/o cycle-equiv.)	≤ 8	≤ 10	786 s	92

■ Some examples which make a *dd-check* necessary are (0/1 stand for complex terms which have been detected in this path previously to be equivalent to 0/1):

- absorption, e.g.,

$$bit_{31} \ \& \ (\text{not } (1 \ \text{nand } (((AK[30] \ \text{and } 1) \ \text{or } 0) \ \text{nand } MI[30]))) \ \text{nand} \\ (0 \ \text{nor } ((AK[30] \ \text{and } 1) \ \text{or } 0)) \ \& \ \dots \ \& \ bit_0 \\ \equiv_{sim} AK$$

- Boolean datapath-operations on bit-vectors, e.g.,

$$bit_{31} \ \& \ 1 \ \text{nand} \ ((1 \ \text{and} \ (MI[30] \ \text{xor} \ AK[30])) \ \text{nor} \ 0) \ \& \ \dots \ \& \ bit_0$$

$$\equiv_{sim} ((\text{vnot} \ AK) \ \text{vand} \ MI) \ \text{vor} \ (AK \ \text{vand} \ (\text{vnot} \ MI))$$
where **vand** etc. are Boolean operations on *bit-vectors*;
- the examples in Fig. 4 and 5.

The verification of problem (4) using *only* vectors of OBDDs at the end of a path *without* evaluating the information of the other equivalence detection techniques as in Tab. 4 ran out of memory.

Verification was automatic, the only user-annotations concern the completion of an instruction for check (2) and (4) and the designation of the 3 (RWA) respectively 5 (MPA) control registers for intermediate *dd-checks*, see section 5.

7 Conclusion and Future Work

Our symbolic simulation approach has been applied to the verification of examples at different levels of abstraction. We have demonstrated that also the *sequential* verification of gate-level- against rtl-descriptions is possible although our examples are still not nearly as complex as commercial designs. Note that verification is independent of the specific commercial synthesis tool and copes also with manual modifications of the designer.

In contrast to previous approaches, the symbolic terms are not modified during simulation, only the information about the corresponding equivalence classes is changed. This permits a flexible use of an open library of different equivalence detection techniques in order to find a good compromise of the accuracy-speed trade-off. An effective combination of symbolic simulation and decision diagrams has been implemented which permits detecting corner-cases of equivalence. An important advantage of the automatic approach is the good debugging support.

Acknowledgement

The author would like to thank the anonymous reviewers and Holger Hinrichsen for helpful comments.

References

- [1] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *DAC'98*, 1998.
- [2] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Formal verification using parametric representations of boolean constraints. In *DAC'99*, 1999.
- [3] P. Ashar, A. Gupta, and S. Malik. Using complete-1-distinguishability for FSM equivalence checking. In *ICCAD'96*, 1996.
- [4] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, volume 1166 of *LNCS*. Springer Verlag, 1996.

- [5] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC'98*, 1998.
- [6] V. Bertacco, M. Damiani, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *DAC'99*, 1999.
- [7] R. E. Bryant. Symbolic verification of MOS circuits. In *1985 Chapel Hill Conference on VLSI*, pages 419–438. Computer Science Press, 1985.
- [8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, 1986.
- [9] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *DAC'90*, 1990.
- [10] M. K. Ganai, A. Aziz, and A. Kuehlmann. Enhancing simulation with BDDs and ATPG. In *DAC'99*, 1999.
- [11] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In *Formal Hardware Verification*, volume 1287 of *LNCS*. Springer Verlag, 1997.
- [12] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufman, CA, second edition, 1996.
- [13] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *DMTCS+CATS'99, Auckland*, 1999.
- [14] S. Höreth. Implementation of a multiple-domain decision diagram package. In *CHARME'97*, 1997.
- [15] S. Höreth. Hybrid graph manipulation package demo, Darmstadt, 1998.
<http://www.rs.e-technik.tu-darmstadt.de/stn/demo.htm>.
- [16] C. N. Ip and D. L. Dill. State reduction using reversible rules. In *DAC'96*, 1996.
- [17] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD'95*, 1995.
- [18] C. Kern and M. R. Greenstreet. Formal hardware verification in hardware design: A survey. *ACM Trans. on Design Automation of Electronic Systems*, 4(2), 1999.
- [19] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, 1999.
- [20] M. Pandey and R. E. Bryant. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. *IEEE Trans. on Computer-Aided Design*, 18(7):918–935, 1999.
- [21] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *CHARME'99*, volume 1703 of *LNCS*. Springer Verlag, 1999.
- [22] G. Ritter, H. Hinrichsen, and H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *ASIAN'99*, volume 1742 of *LNCS*. Springer Verlag, 1999.
- [23] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, 1995.
- [24] T. Uehara. Proofs and synthesis are cooperative approaches for correct circuit designs. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North-Holland, 1987.
- [25] L.-C. Wang, M. S. Abadir, and N. Krishnamurthy. Automatic generation of assertions for formal verification of PowerPCTM microprocessor arrays using symbolic trajectory evaluation. In *DAC'98*, 1998.

Speeding Up Image Computation by Using RTL Information

Christoph Meinel and Christian Stangier

FB Informatik, University of Trier
{meinel,stangier}@uni-trier.de

Abstract. Image computation is the core operation for optimization and formal verification of sequential systems like controllers or protocols. State exploration techniques based on OBDDs use a partitioned representation of the transition relation to keep the OBDD-sizes manageable. This paper presents a new approach that significantly increases the quality of the partitioning of the transition relation of controllers given in the hardware description language Verilog. The heuristic has been successfully applied to reachability analysis and symbolic model checking of real life designs, resulting in a significant reduction both in CPU time and memory consumption.

1 Introduction

The computation of the reachable states (RS) of a sequential circuit is an important task for synthesis, logic optimization and formal verification. The increasing complexity of sequential systems like controllers or protocols requires efficient RS computation methods. If the RS are computed by using Ordered Binary Decision Diagrams (OBDDs) [1], the system under consideration is represented in terms of a transition relation (TR). Since the monolithic representation of the circuit's TR usually leads to unmanageable large OBDD-sizes, the TR has to be partitioned [2, 3]. The quality of the partitioning is crucial for the efficiency of the RS computation. The computation of transitions will be unnecessarily time consuming, if the TR is divided into too many parts. On the other hand a number of partitions that is too small will lead to a blow-up of OBDD-size and hence, memory consumption. Partitioning the TR is usually done without utilizing any external information. The standard method is to sort the latches according to a benefit heuristic [4] and then apply a clustering algorithm. This clustering algorithm follows a greedy scheme [5] that is guided only by OBDD-size, i.e. if the OBDD-size of a partition is exceeding a certain threshold a new partition has to be created.

In this paper we propose a heuristic for partitioning of controllers and protocols that uses information given by the register transfer level (RTL) description of controllers written in Verilog [6]. The application of our heuristic to reachability analysis reduced the computation time by 22% and memory consumption by 15% (overall). Additionally, we performed experiments with symbolic model checking [7]. Here, we got a reduction of CPU time by 67% and memory consumption by 62%, when applying our heuristic. Especially the experiments with model checking, where often properties concerning interaction of functional modules are checked, show the potential of our RTL based approach.

The heuristic is based on but not limited to Verilog. It may easily be adapted to other hardware description languages that provide RTL information like e.g. VHDL [10].

2 Preliminaries

2.1 Verilog

For the purpose of logic synthesis, designs are currently written in a hardware description language (HDL) at register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of *data flow* and *behavioral constructs*. Logic synthesis tools take the RTL HDL description to produce an optimized gate level netlist and high level synthesis tools at the behavioral level output RTL HDL descriptions. Verilog and VHDL are the most popular HDLs used for describing the functionality at RTL. Within the design cycle of optimization and verification the RTL level plays an important and frequently used role.

The design methodology in Verilog is a top down hierarchical modeling concept based on modules, which are the basic building blocks.

2.2 Partitioned Transition Relations

The computation of the RS is a core task for optimization and verification of sequential systems. The essential part of OBDD-based traversal techniques is the transition relation TR:

$$\text{TR}(x, y, e) = \prod_i \delta_i(x, e) \equiv y_i,$$

which is the conjunction of the transition relations of all latches (δ_i denotes the transition function of the i th latch). This *monolithic* TR is represented as a single OBDD and usually much too large to allow an efficient computation of the RS. Sometimes a monolithic TR is too large to be represented with OBDDs. Therefore, more sophisticated RS computation methods make use of a *partitioned* TR [10], i.e. a cluster of OBDDs each of them representing the TR of a group of latches. A transition relation partitioned over sets of latches P_1, \dots, P_j can be described as follows:

$$\text{TR}(x, y, e) = \prod_j \prod_{i \in P_j} \delta_i(x, e) \equiv y_i.$$

The RS computation consists of repeated image computations $\text{Img}(\text{TR}, R)$ of a set of already reached states R :

$$\text{Img}(\text{TR}, R) = \exists_{x,e}(\text{TR}(x, y) \cdot R)$$

With the use of a partitioned TR the image computation can be iterated over P_j and the \exists operation can be applied during the product computation (*early quantification*). The so called *AndExist* [10] or *AndAbstract* operation performs

the AND operation on two functions (here partitions) while simultaneously applying existential quantification ($\exists_x f = f_{x=\text{true}} \vee f_{x=0}$) on a given set of variables, i.e. the variables that are not in the support of the remaining partitions. Unlike the conventional AND operation the AndExist operation does not have a polynomial upper bound for the size of the resulting OBDD, but for many practical applications it prevents a blow-up of OBDD-size during the image computation.

Since the number of quantified variables depends on the order in which the partitions are processed, finding an optimal order of the partitions for the AndExist operation is an important problem. Geist and Beer [4] presented a heuristic for the ordering of partitions each representing a single state variable. A more sophisticated heuristic for partitions with several variables is given by [9].

3 Partitioning of Transition Relations

The quality of the partitioning is crucial for the efficiency of the RS computation. The image computation is iterated over the partitions and includes costly product (i.e. AND) computations. Therefore, maintaining a large number of partitions is time consuming. A small number of partitions may lead to unmanageable large OBDDs. One extremum of this trade-off is the partitioning where each latch forms a partition, which is usually small but requires many iterations. The other extremum is a monolithic TR, that can be computed in one iteration but has large OBDD-size. Furthermore, the order of latches and the clusters is crucial for an efficient AndExist operation.

In the following we will describe the standard partitioning strategy and our new approach.

3.1 Common Partitioning Strategy

A common strategy for partitioning of the TR as it is used e.g. by VIS [1] proceeds in three steps:

1. **Order Latches.** First, the latches are ordered by using a benefit heuristic [2] that performs a structural analysis of the latches transition function to address an effective AndExist operation. Hence, the heuristic considers: variables that may be quantified out, highest index in the function, etc.
2. **Cluster Latches.** The single latch relations are clustered by following a greedy strategy. Latches are added to a OBDD (i.e. by performing AND) until the size of the OBDD exceeds a given threshold.
3. **Order Clusters.** In the last step the clusters are ordered similarly to the latches by using a benefit heuristic (VIS uses the same heuristic as in Step 1).

Figure 1a) gives a schematic overview of this process.

3.2 RTL Based Partitioning Heuristic

The way to build a complex design is to break it into modules, each with a dedicated functionality and a smaller complexity. For example communication protocols contain transmitters and receivers that represent independent modules.

These modules are usually not too complex, thus the complexity of their TRs will be small. If a partition contains state variables of several modules, we need to represent the Cartesian product of these modules leading to a much more complex TR. The main reason for the efficiency of the partitioned TR approach is that state variables not appearing in other partitions are quantified out during the AndExist operation. This leads to much smaller OBDD-sizes and a faster computation. If the state variables of a module are spread over several partitions, the quantification does take effect only late in image computation. Therefore, most of the computation has to be done with large OBDDs.

RTL level description languages like Verilog support a hierarchical design methodology by providing module constructs. As it can be seen this modularization has effects on the image computation (see discussion below) that should not be neglected.

Although the standard method optimizes the partitioning twice, its main disadvantage is that it only uses structural information to optimize the partitioning for an efficient order for the AndExist operation during the image computation.

Our new heuristic improves this optimization by including additional semantical information about the represented functions. As the analysis and the experimental results show, there is a close connection between the RTL description and an efficient image computation.

The RTL heuristic proceeds in three steps:

1. **Group Latches.** The latches are grouped according to the modules given in the top module of the RTL description in Verilog. Within the groups the latches are ordered by a lexicographic order that takes into account submodule names and bit numbers (names of latches from submodules are prefixed by the submodule name). Also, the bits of a certain register are named by the register and the bit number. The effect of this sorting is, that latches of a submodule within the group stay adjacent, without being grouped explicitly. The same holds for the bits of a register.
2. **Cluster Groups.** The groups represent borders for the clusters. There is no cluster containing latches from different groups. To control the OBDD size of the clusters, the greedy partitioning strategy is applied within the groups. The clustering given by the groups lowers the influence of the arbitrary clustering produced by the OBDD-size threshold. Thus, resulting in a more *natural* partitioning.
3. **Order Clusters.** In the last step the clusters may be ordered by using the benefit heuristic from the standard method.

Figure 1 b) gives an overview of this strategy.

Modifications of this strategy are possible:

- **Step 1a)** As an additional step the benefit heuristic of the standard method may be applied to order the latches within the single groups. It emerged that the lexicographic order of the latches preserves more of the structure of the design and leads to better results.
- **Step 2a)** One may allow to create clusters that cross a group border. This will lead to a more compact representation of the TR with fewer clusters. Although the representation is more efficient the image computation does not perform as efficient as with the strict group borders. An explanation for this behavior is given below.

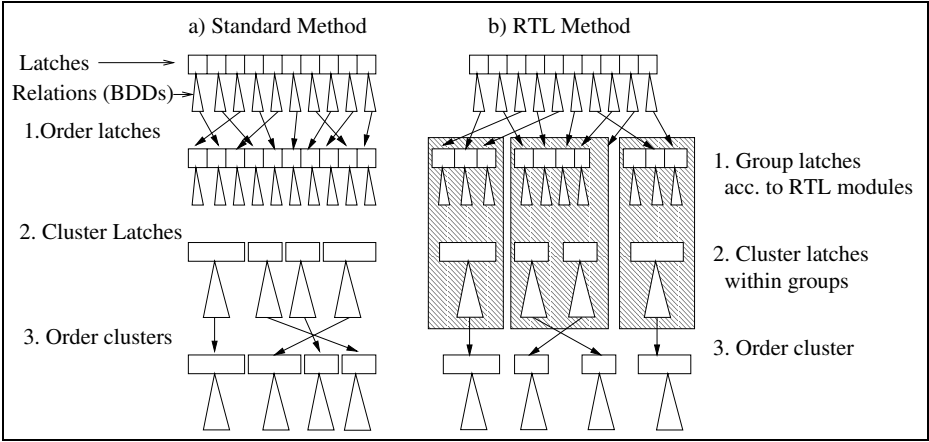


Fig. 1. Schematic of Partitioning Strategies.

3.3 Analysis of the Image Computation

In the following we will discuss the influence of different partitioning schemes on the image computation of controllers that are inherently modularized. For the ease of understanding we consider a hypothetical protocol consisting of a transmitter (Tx) and a receiver (Rx). The state variables of Tx are t_0, \dots, t_m and the corresponding transition functions are $\delta_{t_0}, \dots, \delta_{t_m}$. The state variables of Rx are r_0, \dots, r_n , the corresponding transition functions are $\delta_{r_0}, \dots, \delta_{r_n}$.

A common greedy partitioning strategy merges δ_t s and δ_r s until the threshold for the OBDD-size of the partition is exceeded. We can expect that δ_t s and δ_r s appear in every partition. Hence, every partition depends on all variables $t_0, \dots, t_m, r_0, \dots, r_n$. This kind of “mixed” partitioning will not have negative effects on usual monolithic controllers, but it will have negative effects on modularized controllers:

First, the abstraction operation (\exists) included in the AndExist operation does take effect very late, resulting in large OBDDs in the preceeding computations.

Secondly, if the partitions depend on all variables, the AND operation within the AndExist is a complete AND operation with a worst case complexity of $O(|I_i| \cdot |P_i|)$, where P_i and I_i are the OBDDs of the i th iteration.

Even if the partitions are almost separated, but e.g. δ_{r_0} is represented in a partition of only δ_t s, the OBDD for δ_{r_0} cannot share any nodes with the OBDDs for the other functions. This results in an unnecessary large partition that is again depending on all variables.

We have a different situation if we use a modularized partitioning. Please notice that the set R of already reached states and the resulting Image (Img) of every iteration of the RS computation are independent of the chosen partitioning scheme. Their OBDDs may be different due to variable reordering, but the main reason for the better performance is the different partitioning scheme: Only transition functions of one type (δ_t resp. δ_r) are merged into one partition and iterated consecutively. This scheme performs better for the following reasons:

First, the AND operation is performed on a smaller number of variables. During the i th iteration the partition P_i and the intermediate result I_i share only a fraction of variables. The complexity for the AND operation of two OBDDs A and B with totally disjoint variable sets is $O(|A| + |B|)$. The AND operation in the modularized scheme is much closer to this complexity than the AND operation in the mixed scheme.

Secondly, if the end of a module is reached, the abstract operation will quantify out all variables of this module, resulting in smaller OBDDs.

4 Experiments

4.1 Implementation

We implemented our strategy in the VIS-package [1] (version 1.3) using the underlying CUDD-package [2] (version 2.3.0). VIS is a popular verification and synthesis packages in academic research. It inherits state of the art techniques for OBDD manipulation, image and reachable states computation as well as formal verification techniques. Together with the vl2mv translator VIS provides a Verilog front-end needed for our heuristic.

4.2 Benchmarks

For our experiments we used Verilog designs from the Texas97 benchmark suite [3]. This publicly available benchmark suite contains real life designs including:

- MSI Cache Coherence Protocol
- PCI Local BUS
- PI BUS Protocol
- MESI Cache Coherence Protocol
- MPEG System Decoder
- DLX
- PowerPC 60x Bus Interface

The benchmark suite also contains properties given in CTL formulas for verification.

We chose those designs that represent RTL (i.e. including more than one module) rather than gate level descriptions. Considered were those designs that could be read in and whose transition relation could be build respecting our system limitations. Too small examples (CPU time $< 5s$) were not considered.

4.3 Experimental Setup

We left all parameters of VIS and CUDD unchanged. The most important default values are:

- Partition cluster size = 5000
- Partition method for MDDs = inout
- OBDD variable reordering method = sifting
- First reordering threshold = 4004 nodes

The reachable states computation or the model checking was preceded by a forced variable reordering. The CPU time was limited to 2 CPU hours and memory usage was limited to 200MB. All experiments were performed on Linux PentiumIII 500Mhz workstations.

4.4 Results of Reachability Experiments

As a first experiment we performed reachability analysis on the given benchmarks. For results see Table 1 and Table 2.

RS describes the number of reachable states of the design, Depth its sequential depth. Part gives the number of partitions of the transition relation. The OBDD-size of the transition relation cluster and the peak number of live nodes is given by TRn resp. Peakn. The CPU time is measured in seconds and given as Time. The columns denoted with % describe the improvement in percent ■.

At the bottom of Table 2 you can find the sum of all numbers of partitions, BDD-sizes and CPU-times. Also, the *average of the relative improvement* is given as well as the *total improvement*.

The main result of these experiments is that using the RTL heuristic the reachable states are being computed faster and the OBDD sizes are smaller.

Although the OBDD sizes of the TR are comparable for both methods (the RTL method is 3% smaller), the OBDD peak sizes of the RTL method are 15% smaller than the peak sizes of the standard method. The computation time improves on 22% overall and 19% on average. The heuristic performs worse on 8 benchmarks, but these benchmarks represent only 6% of the original computation time and the time losses for these benchmarks add up to 98 seconds. In most cases the time losses result from extra reordering calls, that are triggered because the OBDDs are smaller (!) in comparison to the standard method.

As a conclusion one may say that the RTL heuristics performs stable and is especially useful for larger designs.

4.5 Results of Model Checking Experiments

In the second series we performed model checking experiments on the basis of the texas97 benchmarks.

In model checking the number of image computations usually is larger than for reachable states computation. On the other hand model checking is not dominated by the size of the reachable states set and the resulting variable reordering effort.

For results see Table 3 and Table 4. The notation in the table is the same as before. Reachable states and the depth of the circuit are not computed. Additionally the number of image computations (img.comp) is given.

The experiments show significant improvements in time and space: The overall CPU time decreased by 67% overall and 40% on average. The RTL method outperforms the standard method for all except two benchmarks. The decrease in computation time ranges up to 90%. The OBDD peak sizes could be lowered by 62% overall and 25% on average. A reason for this remarkable improvement may be the fact that often properties checked by model checking concern the interaction of modules in a design. The RTL heuristic targets exactly this behavior and produces a kind of a *smart* partitioning of the TR with respect to this verification task.

¹ $0 < \text{improvement} < 100$; $-100 < \text{impairment} < 0$

5 Conclusion

We presented a heuristic for optimizing the partitioning of the transition relation for reachable states computation and model checking of sequential systems written in the hardware description language Verilog. The heuristic significantly decreases computation time and memory consumption during reachable states computation and model checking and thus allows more efficient optimization and verification. The heuristic is general enough to be applied to other hardware description languages that provide RTL information.

References

1. A. Aziz et. al., *Texas-97 benchmarks*,
<http://www-cad.eecs.berkeley.edu/Resdep/Research/vis/texas-97/>.
2. R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 1986, pp. 677-691.
3. J. R. Burch, E. M. Clarke, D. E. Long, *Symbolic Model Checking with partitioned transition relations*, Proc. of Int. Conf. on VLSI, 1991.
4. J. R. Burch, E. M. Clarke, D. L. Dill, L. J. Hwang and K. L. McMillan, *Symbolic model checking: 10^{20} states and beyond*, Proc. of LICS, 1990, pp. 428-439.
5. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa, *VIS: A System for Verification and Synthesis*, Proc. of Computer Aided Verification CAV'96, 1996, pp. 428-432.
6. O. Coudert, C. Berthet and J. C. Madre, *Verification of Synchronous Machines using Symbolic Execution*, Proc. of Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, Springer, 1989, pp. 365-373.
7. D. Geist and I. Beer, *Efficient Model Checking by Automated Ordering of Transition Relation Partitions*, Proc. of Computer Aided Verification CAV'94, 1994, pp. 294-310.
8. R. D. M. Hunter and T. T. Johnson, *Introduction to VHDL*, Chapman & Hall, 1996.
9. R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley and B. Plessier, *Efficient BDD Algorithms for Synthesizing and Verifying Finite State Machines*, Proc. of Int. Workshop on Logic Synthesis (IWLS'95), 1995.
10. F. Somenzi, *CUDD: CU Decision Diagram Package*,
<http://vlsi.colorado.edu/fub/>
11. D.E. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991.

Table 1. Comparison of Original VIS Partitioning and RTL Heuristic for RS Computation

	Design						Standard VIS						RTL Method					
	RS	Depth	Parts	TRn	Peakn	Time	Parts	TRn	%	Peakn	%	Time	%					
ONE	2.91e8	16	3	2175	20284	5.32	5	1987	9	19068	6	4.99	6					
PClnorm	2631170	35	23	47472	81543	123.99	25	34646	27	87882	-7	132.75	-7					
PClnorm	86528	30	20	31048	69291	71.93	21	39687	-22	69291	0	81.18	-11					
SDLX.sdlx	12	11	17	26943	38106	75.12	25	12946	52	26324	31	35.23	53					
TEST_PDLX.sdlx	155	154	25	66680	92726	264.74	29	22142	67	51894	44	137.6	48					
TEST_SDLX.sdlx	155	154	25	66680	92726	267.72	29	22142	67	51894	44	137.69	49					
mpeg	2081	17	36	39614	56082	293.66	36	18014	54	41684	26	257.36	12					
multi-main	1144830	41	18	35804	70727	165.1	16	28368	21	49332	30	97.08	41					
p62_LS_LS_V01	2823	61	36	54572	192422	238.25	41	67411	-19	171498	11	245.85	-3					
p62_LS_LS_V02	1045	58	36	54544	135371	112.24	41	67349	-19	117970	13	131	-14					
p62_LS_LS_V01	2743	61	35	78281	193959	294.64	41	72312	8	180578	7	244.58	17					
p62_LS_LS_V02	1124	66	35	78336	122943	129.2	41	71405	9	119839	2	122.11	5					
p62_LS_S_V01	2743	61	35	78281	193959	295.4	41	72312	8	180578	7	247.98	16					
p62_LS_S_V02	1124	66	35	78336	122943	127.86	41	71405	9	119839	2	122.86	4					
p62_LLL_V01	2445	48	36	54572	141902	202.28	41	67399	-19	150146	-5	137.79	32					
p62_LLL_V02	2398	71	36	54544	141716	220.7	41	67349	-19	150533	-6	154.52	30					
p62_LS_S_V01	3637	85	37	73008	149810	234.7	41	73755	-1	158476	-5	231.23	1					

Table 3. Comparison of Original VIS Partitioning and RTL Heuristic for Model Checking

		Standard V/S					RTL Method					
	img.-											
	comp	Peakn	Parts	TRn	Time	Peakn	%	Parts	TRn	%	Time	%
PClabnorm.PCI	304	176276	14	28613	253	145780	17	10	24054	16	157	38
PClnorm.PCI	206	81123	15	35124	56	69291	15	9	24472	30	52	7
TWO contention	37	97623	7	11865	47	168938	-41	10	13207	-9	114	-58
multi:main.mutim	45	38694	5	14700	33	33423	14	4	1588	89	18	45
p62.L.S.LS.V01.ccp	64	166074	23	49952	200	141829	15	22	60455	-16	148	26
p62.L.S.LS.V01.p6live	99	452267	23	49952	827	343308	24	22	60455	-16	367	56
p62.L.S.LS.V02.ccp	53	146494	22	59487	107	121138	17	21	55439	7	65	39
p62.L.S.LS.V02.p6live	96	167454	22	59487	181	139905	16	21	55439	7	120	34
p62.L.S.L.V01.ccp	64	176540	23	49684	216	154852	12	21	61118	-18	141	35
p62.L.S.L.V01.p6live	99	1614200	23	49684	3833	303656	81	21	61118	-18	384	90
p62.L.S.L.V02.ccp	54	148560	23	62140	106	132040	11	22	55605	10	67	37
p62.L.S.L.V02.p6live	89	183811	23	62140	193	136522	26	22	55605	10	118	39
p62.L.S.S.V01.ccp	64	176540	23	49684	211	154852	12	21	61118	-18	136	35
p62.L.S.S.V01.p6live	99	1614200	23	49684	3564	303656	81	21	61118	-18	401	89
p62.L.S.S.V02.ccp	54	148560	23	62140	109	132040	11	22	55605	10	65	40
p62.L.S.S.V02.p6live	89	183811	23	62140	199	136522	26	22	55605	10	114	43
p62.L.L.V01.ccp	52	164244	23	48961	194	142996	13	22	58344	-15	123	37
p62.L.L.V01.p6live	87	477543	23	48961	935	305267	36	22	58344	-15	244	74
p62.L.L.V02.ccp	53	144504	23	48971	180	131449	9	21	55967	-11	73	60
p62.L.L.V02.p6live	96	242452	23	48971	402	234951	3	21	55967	-11	214	47
p62.L.S.V01.ccp	75	168782	22	62479	121	134604	20	22	57345	8	120	1
p62.L.S.V01.p6live	118	192410	22	62479	231	241246	-19	22	57345	8	224	3
p62.L.S.V02.ccp	55	140767	22	57365	108	118826	16	22	51451	10	99	9
p62.L.S.V02.p6live	96	140767	22	57365	112	120239	15	22	51451	10	101	9

Table 4. Comparison of Original VIS Partitioning and RTL Heuristic for Model Checking cont.

	Img.-	Standard VIS					RTL Method						
		comp	Peakn	Parts	TRn	Time	Peakn	%	Parts	TRn	%	Time	%
p62.ND.LS.V01.ccp	83	396642	24	63506	831	342628	14	23	61983	2	603	27	
p62.ND.LS.V02.ccp	103	191386	22	63321	356	166655	13	22	60733	4	230	35	
p62.ND.LS.V02.p6live	192	1564426	22	63321	3922	847880	46	22	60733	4	1339	66	
p62.ND.L.V01.ccp	75	356794	25	65964	824	323997	9	23	58708	11	616	25	
p62.ND.L.V02.ccp	133	5660454	23	60383	7301	1422284	76	23	57198	5	3352	54	
p62.ND.L.V02.p6live	168	5573568	23	60383	7301	990882	82	23	57198	5	1605	78	
p62.ND.S.V02.ccp	84	150630	23	46744	187	157900	-4	23	61745	-23	135	28	
p62.ND.S.V02.p6live	177	645918	23	46744	1221	363854	44	23	61745	-23	402	67	
p62.S.S.V01.ccp	43	147063	23	62209	102	127339	13	21	59643	4	63	38	
p62.S.S.V01.p6live	80	153012	23	62209	107	139405	9	21	59643	4	116	-6	
p62.S.S.V02.ccp	37	129492	23	54800	96	116500	10	21	57223	-3	58	39	
p62.S.S.V02.p6live	74	129492	23	54800	96	116500	10	21	57223	-3	59	38	
p62.V.LS.V01.ccp	108	283494	24	58415	575	229601	19	23	58674	0	353	39	
p62.V.LS.V01.p6live	114	4483034	24	58415	7301	697992	84	23	58674	0	1290	82	
p62.V.LS.V02.ccp	90	165200	23	52073	234	152143	8	21	58544	-10	143	39	
p62.V.LS.V02.p6live	178	1059895	23	52073	2093	776538	27	21	58544	-10	832	60	
p62.V.S.V01.ccp	82	213245	23	61795	258	200514	6	21	48349	22	244	5	
p62.V.S.V01.p6live	127	964988	23	61795	2421	363716	62	21	48349	22	628	74	
p62.V.S.V02.ccp	84	163439	22	54807	200	132192	19	22	56789	-2	134	33	
p62.V.S.V02.p6live	177	351553	22	54807	515	250465	29	22	56789	-2	271	47	
2-proc.bin	264	903917	4	12311	756	176315	80	3	2816	77	133	82	
2-proc.bin.prop2_bin	140	252974	4	11610	154	84942	66	3	3663	68	43	72	
Sum		31384312	950	2374498	49265	12227572	1149	900	2373170	193	16311	1855	
Average of rel. improvements:													
Total improvement:							62%	25%	5%	0%	4%	40%	

Symbolic Checking of Signal-Transition Consistency for Verifying High-Level Designs

Kiyoharu Hamaguchi¹, Hidekazu Urushihara¹, and Toshinobu Kashiwabara¹

Dept. of Informatics and Mathematical Science,
Graduate School of Engineering Science, Osaka University
Toyonaka, Osaka 560-8531, Japan
hama@ics.es.osaka-u.ac.jp

WWW home page: <http://www-kas1.ics.es.osaka-u.ac.jp/hama/indexe.htm>

Abstract. This paper deals with verification of high-level designs, in particular, symbolic comparison of register-transfer-level descriptions and behavioral descriptions. As models of those descriptions, we use state machines extended by quantifier-free first-order logic with equality. Since the signals in the corresponding outputs of such descriptions rarely change simultaneously, we cannot adopt the classical notion of equivalence for state machines. This paper defines a new notion of consistency based on signal-transitions of the corresponding outputs, and proposes an algorithm for checking consistency of those descriptions, up to a limited number of steps from initial states. A simple hardware/software codesign is taken as an example of high-level designs. A C program for digital signal processing called PARCOR filter was compared with its corresponding design given as a register-transfer-level description, which is composed of a VLIW architecture and assembly code. Since this example terminates within approximately 4500 steps, symbolic exploration of a finite number of steps is sufficient to verify the descriptions. Our prototype verifier succeeded in the verification of this example in 31 minutes.

1 Introduction

Recent progress in integrated circuit technology has enabled implementation of very large logic systems on a single chip so that high-level descriptions such as register-transfer-level designs or behavioral designs has come to be given at early stages of design processes. Since, even at those abstraction levels, the complexity of the designs increases year by year, it has become crucial to guarantee the correctness of such high-level designs with some formal verification method. This paper deals with verification of high-level designs, in particular, symbolic comparison of behaviors for register-transfer-level descriptions and behavioral descriptions.

So far, formal verification methods for gate-level descriptions have been widely utilized in processes of circuit designs. In particular, equivalence checking of combinational circuits, and sequential equivalence checking or model checking

of sequential designs [11, 12], have been applied to industrial designs successfully. The verification tasks for high-level designs, however, cannot be handled with those methods in many cases, because high-level descriptions such as those for digital signal processing tend to contain complex arithmetic operations.

To handle such descriptions, we can use quantifier-free first-order logic with equality, or first-order logic with uninterpreted functions. For this logic, equivalence checking of two formulas, or validity checking of a formula has been known to be decidable [13, 14]. In [15], Burch et. al have proposed a verification method for pipelined processors with this logic. Several methods for improving the performance of verification have been contrived, for example, in [16, 17].

Unfortunately, state machines extended with uninterpreted functions can contain infinite number of states, in general. Verification problems relating to such state machines are intractable. For example, model checking has been proved undecidable [18, 19] for such extended state machines. As a result, verifiers including state enumeration such as in [20, 21], are not guaranteed to terminate. Instead, as described later in this section, we adopt symbolic simulation of the extended state machines.

When we obtain some register-transfer-level design from a behavioral design, the timing of data outputs at register-transfer-level varies among designs generally. Furthermore, the behavioral description may not contain any explicit clocks. To compare the behaviors of those descriptions, it is indispensable to devise some criteria to determine equivalence of two high-level descriptions.

Several researchers have proposed methods for comparing two descriptions at different abstraction levels. Devadas and Keutzer [22] proposed a verification method, in which a non-deterministic automaton is generated for representing all possible implementations for a behavioral description, and is compared with an implementation. Bergamaschi and Raje [23] proposed a notion of equivalence for verifying equivalence of behavioral specifications and scheduled implementations. In their approach, correspondence between a specification state and an implementations state is obtained, and values in state variables are compared at each transition point between specification states.

In [24], Genoe et al. proposed a methodology named SFG-Tracing for guaranteeing correctness of transistor-switch-level implementations against high-level behavioral specifications. When the synthesis tool generates lower-level descriptions by partitioning, a mapping function is generated to relate signals at the two levels in space and in time. This mapping function is used to check equivalence of each pair of the corresponding partitions at the two levels. In [25], *flushing* is used to give the correspondence among specification states and implementation states in pipelined microprocessors. In [26], correspondence among states in a behavioral automaton and in a controller automaton is obtained during a synthesis process, and is used for verification.

In the above approaches, correspondence is given among states at two different abstraction levels, and partitions based on those states are generated at each level, so that symbolic simulation of finite steps is sufficient to check the equivalence of each pair of the partitions at the two levels. Such correspondence is, how-

ever, difficult to obtain in some designs. For example, when a pipelined or VLIW microprocessor is used as an implementation, “atomic” operations at a behavioral description are decomposed, and their executions at a register-transfer-level by a microprocessor can overlap. Then, we lose clear correspondence of states in the two levels. In general, as scheduling done by a synthesizer or by hand comes to contain powerful techniques, it becomes hard to find correspondence of states between different abstraction levels.

The approach in this paper requires no correspondence of the states. As a result, however, we cannot partition a description into pieces, for which symbolic simulation of finite steps is sufficient. In this paper, we only consider states within finite steps from initial states, to make the problem tractable. In other words, we check the consistency up to a limited number of finite steps, rather than checking the equivalence of two descriptions completely. Exploring only a finite number of steps limits the usefulness of our approach, but symbolic simulation works efficiently in general, and there are some situations in which possibly this approach works well. For example, in digital signal processing designs, input data are converted to produce output data in a finite number of steps. As a result, usually, it is expected to be sufficient to check behaviors only within a finite number of steps.

Since we do not use any correspondence of states, we need to introduce a new notion of equivalence or consistency to check for two descriptions. In this paper, as for outputs, we focus on their signal transitions. For example, we regard two sequences *ccdddc* and *cddddd* are equivalent, because sequence of the signal transitions $c \rightarrow d$ and $d \rightarrow c$ are the same. As for inputs, in this paper, we assume that the input sequence for an input variable x is stored in a buffer as x^0, x^1, \dots, x^n , and each of the two machines takes the same x^i as their i -th input. This paper proposes an algorithm which takes two state machines extended with uninterpreted functions, and checks the consistency of specified pairs of outputs in terms of signal transitions, up to a given finite steps.

We take a small hardware/software codesign as an example of high-level designs. A short C program for digital signal processing (PARCOR filtering) is compared with its corresponding design given as a register-transfer-level description, which is composed of a VLIW architecture and an assembly program. Only data variables are represented by domain variables of the logic, and the other control variables are represented by boolean bit vectors. Symbolic simulation of a finite number of steps is sufficient to verify this example, because it terminates within approximately 4500 steps. Our prototype verifier succeeded in the verification of this example in 31 minutes.

Section 2 describes an overview of our approach. Section 3 gives relating definitions. Section 4 proposes a main algorithm. Section 5 shows experimental results. Section 6 concludes this paper.

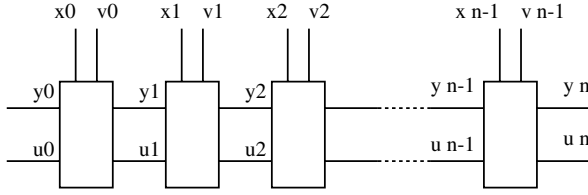


Fig. 1. Expansion of Transition Functions

2 Overview

This section provides an overview of the approach of this paper. Related definitions and an algorithm are given in Section 3 and 4, respectively.

In this paper, high-level descriptions such as register-transfer-level descriptions or behavioral descriptions are modeled by state machine extended by uninterpreted functions, which are simply called *extended state machines*. We assume that a machine is described by a set of state transition functions. For example, the following functions describe an extended state machine.

$$\begin{aligned} Y &:= x \oplus y; \\ U &:= ITE(y, f(u), v); \end{aligned}$$

In addition to a boolean input variable x and a boolean state variable y , we use a *domain* input variable v and a *domain* state variable u . Y and U are next state variables corresponding to y and u respectively. Terms of first-order logic can be assigned to domain state variables. We also give initial values to the variables such as y_0 and u_0 , where y_0 can be a boolean constant or boolean variable, and u_0 can be a *domain constant* or a *domain variable*. In this example, f is a function symbol, that is, an uninterpreted function. The operator *ITE* means *if-then-else*. Thus, the second formula means that, if y is true, then $f(u)$ at the present step is assigned to u at the next step, otherwise the input v at the present step is assigned to u at the next step.

We expand the transition functions of a state machine for a certain number of steps n . An example of the expansion is shown in Figure 1. Input variables are numbered serially for each step. Syntactically, the boolean function or the term representing y^{i+1} or u^{i+1} can be obtained by replacing the boolean state variables or domain state variables with their corresponding boolean functions or terms recursively. We call this procedure *symbolic simulation*.

We assume *observable* state variables, which correspond to outputs. Note that not all of state variables are observable. In the following, we discuss equivalence relating only to observable state variables. For classical state machines, two machines are regarded as equivalent if and only if they produce, in each of corresponding pairs of observable state variables, the same output sequence for every input sequence.

For high-level descriptions, however, this definition of equivalence is not sufficient. To compare a behavioral description with a register-transfer-level description, we are required to regard two descriptions as equivalent, even if a corresponding value does not appear simultaneously in a corresponding pairs of inputs or outputs (i.e., observable variables).

As for inputs, we simply assume that input sequences are all stored in buffers. Since we consider symbolic simulation up to a given limited steps, the input sequence for an input variable x is stored statically as x^0, x^1, \dots, x^n in some state variables, and each of the two machines takes the same x^i as their i -th input.

As for outputs, we focus on signal transitions. Suppose that two state machines have $\sigma_1 = cccddcc$ and $\sigma_2 = cddddccc$ respectively, in observable boolean state variables. When we observe only signal transitions, we can see the sequences in the observable variables $\sigma'_1 = c \rightarrow d \rightarrow c$ and $\sigma'_2 = c \rightarrow d \rightarrow c$. In this case, we regard the sequences as equivalent.

When we perform symbolic simulation of an extended state machine, the sequence of terms corresponding to an observable state variable u , that is, $u^0, u^1, u^2, \dots, u^n$ can represent more than one value under various conditions simultaneously. For example, under some condition c_1 , we have $\sigma_1 = cccddcc$, $\sigma_2 = cddddccc$, but, under another condition c_2 , we have $\sigma_1 = dddccdd$, $\sigma_2 = dddccdd$. We need to regard these sequences as equivalent.

Suppose that we have a formula $F_u(t, s)$ representing a condition under which s -th transition of the value in a variable u occurs at t -th step. Then, the value after s -th transition of values, $value(u, s)$, can be represented as:

$$\begin{aligned} value(u, s) = & \text{ITE}(F_u(1, s), u^1, \\ & \text{ITE}(F_u(2, s), u^2, \\ & \vdots, \\ & \text{ITE}(F_u(n, s), u^n, NIL) \dots)) \end{aligned}$$

where NIL is a special constant, which is used to represent a *value* when s -th change of values does not occur.

Then, for two observable variables u and u' in two machines, if the following formula is valid for every s , then we can say that the signal transitions in u and u' of two machines are consistent up to the n -th step.

$$\begin{aligned} & ((\bigvee_{t=0}^n F_u(t, s)) \wedge (\bigvee_{t=0}^n F_{u'}(t, s))) \\ & \Rightarrow (value(u, s) \doteq value(u', s)), \end{aligned}$$

where the condition $(\bigvee_{t=0}^n F_u(t, s)) \wedge (\bigvee_{t=0}^n F_{u'}(t, s))$ is introduced to guarantee that we consider only cases where s -th transition occurs both in u and u' .

As for the condition $F_u(t, s)$, for example, $F_u(1, 1) = (u^0 \neq u^1)$, $F_u(2, 1) = (u^0 \doteq u^1) \wedge (u^1 \neq u^2)$, $F_u(3, 1) = (u^0 \doteq u^1) \wedge (u^1 \doteq u^2) \wedge (u^2 \neq u^3)$, and so on. The general procedure will be shown in Section 4.

3 The Logic and State Machines

3.1 First-Order Logic with Uninterpreted Functions

Firstly, we define quantifier-free first-order logic with equality, or first-order logic with uninterpreted functions. This logic is used to describe extended state machines.

Syntax. The syntax is composed of terms and formulas. Terms are defined recursively as follows. Domain variables x, y, z, \dots and domain constants a, b, c, \dots are terms. If t_1, t_2, \dots, t_n are all terms, then $f(t_1, t_2, \dots, t_n)$ is a term, where f is a function symbol of order n ($n > 0$). The order means how many arguments can be taken by the function symbol. For a formula α and terms t_1 and t_2 , $ITE(\alpha, t_1, t_2)$ is a term.

Formulas are defined recursively as follows. *true* and *false* are formulas. boolean variables are formulas. $p(t_1, t_2, \dots, t_n)$ is a formula, where p is a predicate symbol of order n ($n > 0$). An equation $t_1 \doteq t_2$ is a formula. If α_1 and α_2 are formulas, then the negation $\neg\alpha_1$ is a formula, and $\alpha_1 \circ \alpha_2$ is also a formula for any binary boolean operator *circ*.

Semantics. For a nonempty domain \mathcal{D} and an interpretation \mathcal{I} of function symbols and predicate symbols, the truth of a formula is defined. The interpretation \mathcal{I} maps a function symbol (resp. predicate symbol) of order k to a function $\mathcal{D}^k \rightarrow \mathcal{D}$ (resp. $\mathcal{D}^k \rightarrow \{\text{true}, \text{false}\}$). Also \mathcal{I} assigns each domain variable and each constant to an element in \mathcal{D} . We restrict the interpretation to constants so that values assigned to distinct constants differ from each other. boolean variables are assigned to $\{\text{true}, \text{false}\}$.

Valuations of a term t and a formula α , denoted by $\mathcal{I}(t)$ and $\mathcal{I}(\alpha)$ respectively, are defined as follows. For a term $t = f(t_1, t_2, \dots, t_n)$, $\mathcal{I}(t) = \mathcal{I}(f)(\mathcal{I}(t_1), \mathcal{I}(t_2), \dots, \mathcal{I}(t_n))$. For a formula $t = ITE(\alpha, t_1, t_2)$, $\mathcal{I}(t) = \mathcal{I}(t_1)$ if $\mathcal{I}(\alpha)$ is *true*, otherwise $\mathcal{I}(t) = \mathcal{I}(t_2)$. For a formula $\alpha = p(t_1, t_2, \dots, t_n)$, $\mathcal{I}(\alpha) = \mathcal{I}(p)(\mathcal{I}(t_1), \mathcal{I}(t_2), \dots, \mathcal{I}(t_n))$. For an equation $\alpha = (t_1 \doteq t_2)$, $\mathcal{I}(\alpha) = \text{true}$ if and only if $\mathcal{I}(t_1) = \mathcal{I}(t_2)$. For $\alpha = \neg\alpha_1$ or $\alpha = \alpha_1 \circ \alpha_2$ for a boolean operator \circ , $\mathcal{I}(\alpha) = \neg\mathcal{I}(\alpha_1)$ and $\mathcal{I}(\alpha) = \mathcal{I}(\alpha_1) \circ \mathcal{I}(\alpha_2)$ respectively.

Validity. A formula α is valid if and only if $\mathcal{I}(\alpha)$ is true for any interpretation \mathcal{I} and any domain \mathcal{D} .

3.2 Extended State Machines

An extended state machine is defined by a set of transition functions. To describe transition functions, we assume four types of variables: boolean input variables x_1, x_2, \dots, x_q , boolean state variables y_1, y_2, \dots, y_r , domain input variables v_1, v_2, \dots, v_s and domain state variables u_1, u_2, \dots, u_t . We introduce next

state variables Y_1, Y_2, \dots, Y_r and U_1, U_2, \dots, U_t corresponding to y_1, y_2, \dots, y_r and u_1, u_2, \dots, u_t respectively. We also describe x_1, x_2, \dots, x_q by \mathbf{x} and so on. Transition functions are described by either of $Y_i := \alpha_i$ or $U_i := t_i$, where α_i and t_i are a formula and a term of the first-order logic with uninterpreted functions, respectively, whose variables are $\mathbf{x}, \mathbf{y}, \mathbf{Y}$ and \mathbf{U} . Boolean or domain constants can also appear in the formula or the term. Furthermore, we assume an initial state. An initial state is an assignment of a boolean value $c_i \in \{true, false\}$ or a boolean variable y_i^0 to each boolean state variable y_i , and an assignment of a domain constant d_j or a domain variable u_j^0 to each domain state variable u_j .

For a usual finite state machine, its transition functions can be described by boolean formulas. In this case, the behavior of the machine depends on an initial state and a sequence of boolean assignments to input variables for each step. Similarly, for an extended state machine, the behavior of the machine depends on an initial state and a sequence of interpretations for each step. We consider a *sequential interpretation* $\mathcal{I} = \mathcal{I}_0, \mathcal{I}_1, \dots$, where \mathcal{I}_i is an interpretation over a common domain \mathcal{D} . To maintain consistency, the sequential interpretation for an extended state machine must satisfy the following conditions.

- The interpretation of constants, function symbols and predicate symbols are the same at every step. In other words, for each boolean constant c_j and each domain constant d_j ,

$$\mathcal{I}_0(c_j) = \mathcal{I}_i(c_j) \text{ for all } i \text{ and } \mathcal{I}_0(d_j) = \mathcal{I}_i(d_j) \text{ for all } i,$$

and also, for each function symbol f and each predicate symbol p ,

$$\mathcal{I}_0(f) = \mathcal{I}_i(f) \text{ for all } i \text{ and } \mathcal{I}_0(p) = \mathcal{I}_i(p) \text{ for all } i.$$

- The values of state variables at a step are determined by the values of state variables and input variables in the previous step. Thus, for transition functions $Y_j := \alpha_j$ and $U_j := t_j$, and for $i = 0, 1, \dots$,

$$\mathcal{I}_{i+1}(y_j) = \mathcal{I}_i(\alpha_j) \text{ and } \mathcal{I}_{i+1}(v_j) = \mathcal{I}_i(t_j).$$

- For an initial state in which c_j and d_j are assigned to y_j and v_j respectively, the following must hold:

$$\mathcal{I}_0(y_j) = \mathcal{I}_0(c_j) \text{ and } \mathcal{I}_0(v_j) = \mathcal{I}_0(d_j).$$

The behavior of an extended state machine is characterized by the set of the sequential interpretations which satisfy the above conditions. We call a sequential interpretation which satisfies the above conditions a *normal sequential interpretation*.

The boolean function or the term representing y^{i+1} or v^{i+1} can be obtained by replacing the boolean state variables or domain state variables with their corresponding boolean functions or terms recursively. We call this procedure *symbolic simulation*.

3.3 Equivalence and Consistency Based on Signal-Transitions

We consider an observable state variable v of an extended state machine. Given a normal sequential interpretation \mathcal{I} , the sequence of values which appear in v is $\sigma_v = \mathcal{I}_0(v)\mathcal{I}_1(v)\mathcal{I}_2(v)\cdots$. The *contraction* of σ_v , $Cont(\sigma_v) = s_1s_2s_3\cdots$, is the sequence which is composed of the values after changes of values in v , which is formally defined as the sequence $s_1s_2s_3\cdots$ which satisfies the following:

- $s_1 = \mathcal{I}_{i_1}(v)$ if and only if $\mathcal{I}_0(v) = \mathcal{I}_1(v) = \cdots = \mathcal{I}_{i_1-1}(v)$ and $\mathcal{I}_{i_1-1}(v) \neq \mathcal{I}_{i_1}(v)$.
- $s_j = \mathcal{I}_{i_j}(v)$ for $j = 2, 3, \dots$ if and only if $s_{j-1} = \mathcal{I}_{i_{j-1}}(v)$ and $\mathcal{I}_{i_{j-1}+1}(v) = \mathcal{I}_{i_{j-1}+2}(v) = \cdots = \mathcal{I}_{i_j-1}(v)$ and $\mathcal{I}_{i_j-1}(v) \neq \mathcal{I}_{i_j}(v)$.

Note that, even if σ_v is an infinite sequence, $Cont(\sigma_v)$ can be a finite sequence or an empty sequence. For example, if $\mathcal{I}_i(v)$ has the same value for all i 's, then $Cont(\sigma_v)$ is an empty sequence.

Let us consider two observable state variables v and v' of a state machine. If $Cont(\sigma_v) = Cont(\sigma_{v'})$ for any normal sequential interpretation, then the sequences in v and v' are said to be *equivalent in terms of signal-transitions*, or simply *equivalent*. Let us describe finite prefixes of σ_v and $\sigma_{v'}$ of length n as ρ_v and $\rho_{v'}$, respectively. If $Cont(\rho_v)$ is a subsequence of $Cont(\rho_{v'})$, or $Cont(\rho_{v'})$ is a subsequence of $Cont(\rho_v)$ for any normal sequential interpretation, then the sequences in v and v' are said to be *consistent in terms of signal-transitions*, or simply *consistent* up to n steps.

When we compare two observable state variables of two state machines, we compose a product machine to check the equivalence or consistency in terms of signal-transitions. In the following, we handle only consistency of observable state variables of two state machines, up to a given number of steps.

4 An Algorithm

Suppose that we have performed symbolic simulation up to n steps, for an observable state variable v . We assume that inputs up to n steps are given in a subset of variables corresponding to initial states. Then, the term representing the value in v at the i -th step, v^i , is composed of only variables corresponding to initial states.

In the following, we construct four types of formulas for v : $change(t)$, $unchange(t)$, $G(t, s)$ and $F(t, s)$. Their meanings are:

- $change(t)$ for $t = 1, 2, \dots, n$: This formula holds if and only if the value of v at t -th step is different from that at $t - 1$ -th step. This is constructed as:

$$v^{t-1} \neq v^t$$

- $unchange(t)$ for $t = 1, 2, \dots, n$: This formula holds if and only if the value of v at t -th step is the same as that at $t - 1$ -th step. This is constructed as:

$$v^{t-1} = v^t$$

- $G(t, s)$ for $t = 0, 1, 2, \dots, n$ and $s = 0, 1, 2, \dots, n$: This formula holds if and only if the total number of transitions of values in v up to t -th step is equal to s .
- $F(t, s)$ for $t = 1, 2, \dots, n$ and $s = 0, 1, 2, \dots, n$: This formula holds if and only if the s -th transition of values in v occurs at the t -th step.

The formula $G(t, s)$ is constructed recursively as follows:

$$G(0, 0) = \text{true}$$

$$G(t, 0) = \bigwedge_{i=1}^t \text{unchange}(i) \quad (t \neq 0)$$

$$G(t, s) = (G(t-1, s-1) \wedge \text{change}(t)) \vee (G(t-1, s) \wedge \text{unchange}(t))$$

Then, we can also construct $F(t, s)$ as:

$$F(t, s) = G(t-1, s-1) \wedge \text{change}(t)$$

As we have seen in Section 2, the value after s -th transition of values, $\text{value}(v, s)$, can be represented as:

$$\begin{aligned} \text{value}(v, s) = & \text{ITE}(F_v(1, s), v^1, \\ & \text{ITE}(F_v(2, s), v^2, \\ & \vdots \\ & \text{ITE}(F_v(n, s), v^n, \text{NIL}) \dots) \end{aligned}$$

where NIL is a special constant, which is used to represent a *value* when s -th change of values does not occur.

Then, the problem of checking consistency up to n steps, for observable state variables v and v' of two machines, is reduced to the validity checking of the following formula for every s :

$$F_{\text{consistency}} = ((\bigvee_{t=0}^n F_v(t, s)) \wedge (\bigvee_{t=0}^n F_{v'}(t, s))) \Rightarrow (\text{value}(v, s) \doteq \text{value}(v', s)),$$

The condition $(\bigvee_{t=0}^n F_u(t, s)) \wedge (\bigvee_{t=0}^n F_{u'}(t, s))$ is introduced to guarantee that we consider only cases where s -th transition occurs both in u and u' .

Positive Equality

Heuristics using *positive equality* proposed in [4], can reduce the complexity of validity checking drastically. This heuristics is based on the property such that, if domain variables or functions are not under negation of equality, then we have only to consider distinct instantiations for those variables or functions. Even if simple equality checking between terms corresponding to observable variables v and v' can exploit this property, however, the construction of $F(t, s)$ introduces negation of equality such as $v^{t-1} \neq v^t$,

Still we can say that the verification result is correct, if the formula is determined to be valid with this heuristics. In other words, we can have only false negative results. Suppose that the heuristics can be applied to any simple equality checking of pair of v^i and v'^i . Let us apply the heuristics to the validity checking of $F_{consistency}$. The heuristics considers only cases where related domain variables or functions have distinct values. Intuitively, this makes more of negations of equality such as $v^{t-1} \neq v^t$ hold. Note that the formulas of this form are used only in conditions for identifying the steps in which signal transitions occur. This means that the heuristics based on positive equality can introduce possibly redundant steps in which values of v and v' are compared. If v and v' are proved to be equal at all of those steps including the redundant steps, we can conclude that the verification result is correct.

5 Implementation and Experiments

We have implemented a prototype verifier based on the procedure shown in the earlier sections. This verifier takes a set of transition functions of an extended state machine (or a product machine) with a number of steps to be explored, and a set of pairs of observable state variables to be checked. In order to perform validity checking of the formula, we have also implemented a validity checker for first-order logic with uninterpreted functions.

As an algorithm for validity checking of $F_{consistency}$, we used a slight modification of an algorithm called boolean method proposed in [1], where OBDDs are used for handling boolean formulas. We also used the heuristics exploiting positive equality. The algorithm in [2] converts a formula including uninterpreted functions to a boolean formula or OBDDs. Since this produces intractably large formulas for our example, our verifier converts a formula with uninterpreted functions, that is, the descriptions of transition functions to OBDDs at each step dynamically to obtain OBDDs representing terms v_i at all steps to be considered. Then, the verifier constructs $F_{consistency}$ to check its validity.

To see feasibility of the algorithm, we took a hardware/software codesign in [3] as an example, and performed consistency checking of a behavioral description written in C, and a register-transfer level description including a VLIW processor and assembly code. This example computes coefficients for a PARCOR filter. The behavioral description shown in Figure 4 computes coefficients from data given in the array x and writes results to the array k .

The corresponding extended state machine was written *manually* from this description, by inserting states between lines of the C code. Only data variables in arrays x , b , n , d or k are expressed by domain variables. Control variables such as i or nn are expressed by boolean bit vectors. The numerical constants were converted to domain constants. The sizes of arrays are 10 for b , n , d and k . The number of data in x is specified by 4-bit boolean vectors j . This means that the verification result covers all the possible cases for j .

The VLIW processor has pipelines of two stages, and can issue 4 instructions simultaneously. The instruction set is similar to those of DLX [4]. It contains 8


```

#define N_DATA j /* 4-bit boolean vector */
#define N_DIM 10

int x[N_DATA];
int b[N_DIM], n[N_DIM], d[N_DIM], k[N_DIM];
int a;

#include "parcor.x"

main()
{
    int i, nn, m;
    int fn, b1, b2;

    a = 0x7eab; /* 0.98958 */
    x[0] = 0;

    for(i = 0; i < N_DIM; i++) {
        /* S1 */
        b[i] = 0;
        k[i] = 0;
        n[i] = 0;
        d[i] = 0x7ffff; /* 1.0 */
    }

    for(nn = 1; nn < N_DATA; nn++) { /* S2 */
        /* S3 */
        fn = x[nn] - x[nn - 1];
        b1 = fn;
        for(m = 0; m < N_DIM; m++) { /* S4 */
            b2 = b[m];
            /* S5 */
            b[m] = b1;
            n[m] = n[m] * a + fn * b2;
            d[m] = d[m] * a + fn * fn;
            b1 = b2 - fn * k[m];
            fn = fn - b2 * k[m];
            /* S6 */
            k[m] = n[m] / d[m];
            /* S7 */
        }
    }

    return;
}

```

Fig. 2. PARCOR filter program[7]

```

_main:
; Function 'main'; 80 bytes of locals, 0 regs to save.
((li r1,_a)(li r3,#1048320)(li r4,#1048320))
((li r6,#32427)(sw -8(r4),r2)(sw -4(r4),r3))
((li r1,_x)(sw (r1),r6))
((li r7,#0)(li r2,#0))
((li r5,_b)(sw (r1),r7)(subi r4,r4,#88)(add r3,r0,r4))
L5:
((li r7,_k)(li r6,#0))
((li r7,_n)(add r1,r2,r7)(sw (r5),r6))
((li r7,_d)(add r1,r2,r7)(sw (r1),r6))
((add r1,r2,r7)(sw (r1),r6))
((addi r2,r2,#4)(li r6,#524287))
((slei r1,r2,#36)(sw (r1),r6))
((bnez r1,L5)(addi r5,r5,#4))
(())
((li r6,_x)(li r7,#1))
((addi r7,r6,#4)(sw -60(r3),r6)(sw -12(r3),r7))
((sw -68(r3),r7))
L10:
((lw r7,-60(r3))(lw r6,-68(r3)))
((lw r1,(r7))(lw r2,(r6)))
((sub r2,r2,r1))
((li r6,#0)(lw r2,-20(r3))(sw -20(r3),r2))
((sw -52(r3),r6))
L14:
((li r6,_b)(lw r7,-52(r3)))
((add r1,r7,r6))
((lw r7,(r1)))
((lw r6,-52(r3))(li r7,_n)(sw -28(r3),r7)) # -28(r3)= b2
((add r7,r6,r7))
((li r1,_a)(sw -36(r3),r7)(sw (r1),r2)) # (sw (r1),r2): b[m] = b1
((lw r7,-28(r3))(lw r6,-20(r3))(lw r1,(r1))(lw r2,(r7)))
((mul r1,r6,r7)(mul r2,r2,r1)(sw -44(r3),r1))

```

Fig. 3. Assembly code for the PARCOR filter program[7]

registers (r0 - r7). The assembly code is shown in Figure 4 and 5. This code was also manually converted to the input for the verifier. The memory necessary to store data or instructions is modeled by a finite number of domain variables or boolean bit vectors.

From a practical point of view, the manual translation is unacceptable, because it takes too much of time and it can introduce new design errors. We expect, however, that it would not be difficult to automate this procedure by some algorithms.

A register can be loaded both with the indices of the arrays expressed by boolean bit vectors, and the data expressed by domain variables. This is not allowed syntactically. The validity checking algorithm, however, handles domain

```

((lw r6,-36(r3))(add r1,r2,r1))
((li r6,_d)(lw r7,-52(r3))(sw (r6),r1))
((add r5,r7,r6))
((lw r7,-44(r3))(lw r1,(r5)))
((lw r7,-20(r3))(lw r6,-20(r3))(mul r2,r1,r7))
((mul r1,r6,r7))
((li r7,_k)(lw r6,-52(r3))(add r2,r2,r1))
((add r5,r6,r7)(sw (r5),r2))
((lw r6,-20(r3))(lw r2,(r5))(sw -84(r3),r2))
((lw r7,-36(r3))(mul r6,r6,r2))
((lw r6,-84(r3))(lw r1,(r7))(sw -76(r3),r6))
((lw r6,-52(r3))(div r1,r1,r6))
((addi r6,r6,#4))
((lw r6,-76(r3))(sw -52(r3),r6)(lw r7,-28(r3)))
((lw r7,-20(r3))(sub r2,r7,r6)(mul r1,r7,r2)(sw (r5),r1))
((sub r7,r7,r1)) #r7 = fn #r2 = b1
((lw r6,-52(r3))(sw -20(r3),r7))
((slei r1,r6,#36))
((bnez r1,L14))
(( ))
((lw r7,-60(r3)))
((addi r7,r7,#4))
((lw r7,-12(r3))(sw -60(r3),r7))
((addi r7,r7,#1)(lw r6,-68(r3)))
((slei r1,r7,j)(addi r6,r6,#4))
((bnez r1,L10)(sw -12(r3),r7)(sw -68(r3),r6))
(( ))
((lw r1,-44(r3)))
((jal _exit))
(( ))
.endproc _main

```

Fig. 4. Assembly code for the PARCOR filter program (cont.)[7]

variables as boolean bit vectors internally. Thus, the implemented verifier can handle this abuse properly, by 'overloading' them into the same width of boolean bit vectors.

We ran the program on a machine with Pentium II (450MHz,1GByte). This example terminates within approximately 4500 steps. Taken $k[0]$ to $k[9]$ as observable state variables, the prototype verifier succeeded in the verification in 31 minutes. Note that the state variables other than $k[0]$ to $k[9]$ were not compared. Symbolic simulation of the state machine dominated almost all the CPU time, and the rest of the task finished in less than 1 minute.

6 Concluding Remarks

This paper proposes an algorithm which takes two state machines extended with uninterpreted functions, and checks the consistency of specified pairs of outputs in terms of signal transitions, up to a given finite steps. Presently, the performance of symbolic simulation has limited the scale of tractable designs. When a function $f(x)$ appears at a step, the algorithm for symbolic simulation checks all the occurrences of form $f(x')$ at earlier steps to check whether $f(x)$ should be regarded as equivalent to $f(x')$. This procedure requires more of computational time, as the number of steps to be simulated increases. Intuitively, it is not necessary to keep all of those possibly 'expired' data of earlier steps. By reducing the range to be managed appropriately, the performance is expected to be improved. Similarly, the algorithm for checking consistency could be modified so that it manages limited local ranges in symbolic simulation. By doing this, it is expected to come to handle larger examples.

Acknowledgments

We gratefully acknowledge Prof. Masaharu Imai and Prof. Yoshinori Takeuchi of Osaka University, Prof. Jun Sato of Tsuruoka National College of Technology and Mr. Norimasa Ohtsuki of Hitachi, Ltd. for valuable discussions and for the PARCOR filter design they kindly allowed us to use in this paper.

References

1. R. A. Bergamaschi and S. Raje. Observable Time Windows: Verifying The Results of High-Level Synthesis. *IEEE Design and Test of Computers*, 14(2):40–50, 1997.
2. R. E. Bryant, S. German, and M. N. Velve. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Conference on Computer-Aided Verification, LNCS 1633*, pages 470–482, July 1999.
3. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.
4. J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of International Conference on Computer-Aided Design*, pages 68–80, June 1994.
5. O. Coudert, C. Berthet, and J.-C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, November 1990.
6. D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In *Conference on Computer-Aided Verification, LNCS 818*, pages 247–259, July 1994.
7. S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proc. of ICCAD*, pages 30–33, 1990.
8. M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man. Illustration of the SFG-Tracing Multi-Level Behavioral Verification Methodology, by the Correctness Proof of a High to Low Level Synthesis Application in CATHEDRAL-II. In *Proc. of IEEE International Conference on Computers and Design*, pages 338–341, 1991.

9. M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man. Automatic Formal Verification of Cathedral-II Circuits from Transistor Switch Level Implementations up to High Level Behavioral Specifications by the SFG-Tracing Methodology. In *Proc. of European Conference on Design Automation (EDAC-92)*, pages 16–19, 1992.
10. J. L. Henny and D. A. Paterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
11. R. Hojati, A. Isles, D. Kirkpatrick, and R. K. Brayton. Verification using uninterpreted functions and finite instantiations. In *Formal Methods in Computer-Aided Design, LNCS 1166*, pages 218–232, November 1996.
12. M. Imai, Y. Takeuchi, N. Ohtsuki, and N. Hikichi. Compiler Generation Techniques for Embedded Processors and Their Application to HW/SW Codesign. In Ahmed A. Jerraya and Jean Mermet, editors, *System-Level Synthesis*, pages 293–320. Kluwer Academic Publishers, 1999.
13. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient Validity Checking for Processor Verification. In *Proceedings of International Conference on Computer-Aided Design*, pages 2–6, November 1995.
14. N. Mansouri and R. Vemuri. A Methodology for Automated Verification of Synthesized RTL Designs and Its Integration with a High-Level Synthesis Tool. In *Formal Methods in Computer-Aided Design (FMCAD 98)*, pages 204–221, 1998.
15. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedure. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, 1979.
16. R. E. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of ACM*, 26(2):351–360, 1979.
17. Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In *Conference on Computer-Aided Verification, LNCS 1427*, pages 219–231, July 1998.

Symbolic Simulation with Approximate Values^{*}

Chris Wilson¹, David L. Dill¹, and Randal E. Bryant²

¹ Computer Systems Laboratory, Stanford University, Stanford, CA 94035
`chriswi@stanford.edu`, `dill@cs.stanford.edu`

² Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213
`Randy.Bryant@cs.cmu.edu`

Abstract. Symbolic methods such as model checking using binary decision diagrams (BDDs) have had limited success in verifying large designs because BDD sizes regularly exceed memory capacity. Symbolic simulation is a method that controls BDD size by allowing the user to specify the number of symbolic variables in a test. However, BDDs still may blow up when using symbolic simulation in large designs with a large number of symbolic variables. This paper describes techniques for limiting the size of the internal representation of values in symbolic simulation no matter how many symbolic variables are present. The basic idea is to use approximate values on internal nodes; an approximate value is one that consists of combinations of the values 0, 1, and X . If an internal node is known not to affect the functionality being tested, then the simulator can output a value of X for this node, reducing the amount of time and memory required to represent the value of this node. Our algorithm uses categorization of the symbolic input variables to determine which node values can be more approximate and which can be more exact.

1 Introduction

Verification methods can be categorized into two basic types: full and partial. Full methods attempt to verify all functionality in one shot while partial methods attempt to verify functionality in pieces. Formal verification attempts to verify functionality fully using symbolic methods. Model checking is a formal verification method that works well on small designs, but does not scale to larger designs because the amount of memory required often exceeds capacity on larger designs. Simulation based methods such as directed and random testing scale easily to large designs but require significant test development effort to cover all functionality.

One approach to improving verification is to use symbolic simulation. In our application, symbolic simulation is a partial verification method that extends directed tests by using symbolic values on inputs. This allows exploring more functionality than is possible with a single directed or random test. We call

^{*} This work is supported by the MARCO/DARPA Gigascale Silicon Research Center (GSRC). We also thank HAL Computer Systems for the use of their designs and resources.

symbolic simulation used in this way *symbolic system simulation* to distinguish it from other forms of symbolic simulation.

This paper addresses the problem of using symbolic system simulation on designs that include both data and control logic. In particular, we are concentrating on the verification of system-level designs. A system-level design is one that integrates a number of units that include both datapath and control logic. The goal of system-level testing is primarily to verify the interaction between units rather than exhaustively verify the functionality of each unit.

One of the characteristics of doing partial verification of large system-level designs is that many system inputs are don't care inputs during a test. A *don't care node* is a node in the circuit whose output value should not affect the outcome of the test; a *don't care input* is a don't care node that is a primary input to the circuit. A *care node* is the opposite of a don't care node.

Conventional symbolic simulation using BDDs does not handle don't care inputs well. For example, suppose a circuit has a multiplier that is not being used for a particular test. The user would like to put symbolic don't care variables on the multiplier inputs. However, BDDs are known to blow up for multipliers and it is likely that the BDDs for the multiplier will cause memory overflow. But, since the multiplier is unused for this test, memory overflow causes the test to abort unnecessarily.

Our solution to this problem is to use *approximate values* on internal nodes that limit the amount of memory blow-up. An approximate value is one that can have the value 0, 1, and X as a function of the symbolic variables in the test. Using approximate values allows the test to complete no matter what kind of don't care logic is present in the design. If the simulator knows that a particular node in the circuit is a don't care node, it can output an X value for this node. For care nodes, it can compute the exact value represented by a BDD for the node since it knows this value will affect the output.

The basic issue in implementing this is: How does the simulator distinguish between care and don't care nodes? Our method works by categorizing symbolic input values as care or don't care values and then creating more exact or more approximate values on nodes based on this categorization. The result of this is that the amount of approximation needed at each node can be determined by looking at the input values to that node only.

An additional problem in symbolic system simulation is that BDDs may overflow memory for functionality that the user wants to verify. For example, suppose we were trying to verify a multiplier as part of a larger circuit. In system simulation, usually what is being verified is that the multiplier is correctly hooked up and communicates properly with the rest of the chip. In this case, producing some result is more important than exhaustively verifying the entire multiplier. Exhaustive verification of the multiplier can be done in a stand-alone unit environment using methods more suitable for that.

Our simulator handles BDD overflow using techniques borrowed from satisfiability (SAT) checking. In particular, our algorithm is closely related to the Davis-Putnam algorithm [6]. The algorithm uses recursive case splitting to re-

duce the size of BDDs while maintaining completeness of the verification. Case splitting consists of picking some symbolic input variable and alternately setting it to the constant 0 and re-simulating the circuit and then setting it to the constant 1 and re-simulating.

Case splitting reduces BDD size by converting variables to constants, but increases simulation time due to re-simulation. Recursive case splitting allows the simulator to always produce some result no matter how many care variables are being used since it will turn as many variables as necessary into constants to reduce the BDD size to fit in memory. At the same time, there is no loss of coverage because the simulator enumerates all combinations of values of variables that were split. Practically speaking, completeness is not guaranteed since the number of combinations that needs to be enumerated is exponential in the number of variables split. However, in the event that the user terminates the simulation before all combinations are covered, each combination that has been simulated provides some amount of coverage. This often is sufficient to verify that, for example, the multiplier above is correctly communicating with the rest of the circuit.

2 Related Work

The symbolic simulation methodology most closely related to ours is Symbolic Trajectory Evaluation (STE) [15]. STE encodes sets of ternary vectors as pairs of BDDs which are then propagated through the simulator. The only chance for approximation in this method is in the selection of the ternary vectors, which is done by the user. Our methodology allows the simulator to choose the amount of approximation at each internal node. STE works more efficiently in verifying small datapath units than our method since there is no advantage to approximating values if there are no don't care inputs. However, our method still works well on these cases, whereas symbolic simulation without approximation does not work well on large system-level designs with many don't cares.

A system that is very similar to STE is the commercial symbolic simulator from Innologic. Innologic's simulator is BDD based, but has the ability to handle BDD overflow. Their overflow handling algorithm is not known to us, however.

Some attempts have been made at minimizing BDD overflow in symbolic simulation. Parametric forms [10,11] are a method of encoding the input space of a symbolic test to reduce the size of BDDs in the simulator. This requires the user to determine how to do this and the simulator has no choice in how to evaluate each node. Another method described by Bertacco et. al. [1] uses dependencies between nodes to reduce the size of BDDs at nodes. This requires non-local knowledge of the circuit to compute values at each node and so it is not clear how scalable this method is.

Our use of SAT methods most closely resembles non-clausal satisfiability checking algorithms. Our method directly incorporates the SAT decision procedure into the simulator. Other verification methods that use SAT methods typically work by generating a clausal formula that is fed to an off-the-shelf SAT

checker. An example of this is Bounded Model Checking (BMC) [2]. The problem with using clausal SAT methods is that they require the circuit to be unrolled for however many cycles are being simulated, requiring memory proportional to the product of design size and the number of cycles being unrolled. Our method does not require unrolling the circuit and so allows larger circuits to be simulated over more cycles.

Approximation is widely used in model checking. There are basically two types used. In one method, an exact model is used, but the state space is approximated to keep BDD sizes down [4,8]. Another way is to abstract the model itself [9,13,7]. The latter method is often used to extract tests for simulation to increase state coverage. These methods all have the problem that they do not scale to large designs easily, they require a lot of work and expertise, and the abstraction often hides many bugs.

Another model checking method uses liberal abstraction to handle BDD overflow. One method [14] tries to find the closest subset to the original function. Liberal abstraction is useful in model checking since it simply reduces the state space searched. However, in symbolic simulation, liberal abstraction results in the wrong answer being produced and it is not clear how to compensate for this. Also, for don't care logic, which comprises most of the values in a simulation run, outputting any other approximation than X is a waste of time, so these methods are probably inefficient compared to our method.

3 Background

3.1 Simulation and Symbolic Tests

The input to the simulation process consists of a circuit and a test which specifies some functionality that needs to be verified and how to test it. A circuit consists of a network of nodes. This paper assumes nodes are either two input AND gates, two input OR gates, NOT gates, or primary inputs and outputs.

A symbolic test performs the following basic actions: creates symbolic variables, sets values from the symbolic domain on inputs, simulates the circuit, and checks outputs against expected values. Although a single test may check many outputs, all of these results are combined to give a final *fail* output. A value of 1 at the *fail* output indicates the existence of a bug and the test fails. A 0 at the *fail* output indicates the test passed and there are no bugs in the functionality being tested.

3.2 Ternary Valued Simulation

Let $\mathcal{T} = \{0, 1, X\}$ be the ternary domain of values that can appear on nodes in the circuit. The value X denotes the fact that the actual value could be 0, 1, or some combination of 0 and 1, but that the simulator does not know or does not care about the actual value.

We form the upper semi-lattice $\langle \mathcal{T}, \sqsubseteq \rangle$ defined as $1 \sqsubseteq X$, $0 \sqsubseteq X$, and $a \sqsubseteq a$ for all $a \in \mathcal{T}$. The functions AND, OR, and NOT are defined over this semi-lattice.

In order for the simulator to be sound, these functions must be monotonic, that is the relationship:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \tag{1}$$

must hold. Table 1 shows a monotonic implementation of the AND function over the ternary domain.

Table 1. Table for AND

\wedge	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

Ternary simulation is performed by evaluating each node in the circuit using the ternary extension of the Boolean operation defined for each node over the input values at that node.

3.3 Symbolic Simulation

Let \mathcal{V} be the set of all the symbolic variables in a test. A *literal* is a variable or its complement. An assignment to \mathcal{V} is a function $\phi: \mathcal{V} \rightarrow \{0, 1\}$ that maps variables to Boolean values. Let Φ be the set of all possible assignments. The value of a node in the circuit is a function $f: \Phi \rightarrow \{0, 1, X\}$ that maps each assignment in Φ to a ternary value. This function is called the *value function* of the node, which is not to be confused with the operation function (AND, OR, NOT) for that node. Each node in the circuit has its own value and consequently, its own value function.

We pre-define some value functions that will be useful later. For each variable $a \in \mathcal{V}$, let \hat{a} be the value function defined such that $\hat{a}(\phi) = \phi(a)$. Let $\hat{0}$, $\hat{1}$, and \hat{X} be those value functions that return the values 0, 1, and X respectively for all assignments.

Symbolic simulation consists of computing an output value function for each node given value functions for the input nodes. The computation is done point-wise:

$$(f \langle op \rangle g)(\phi) = f(\phi) \langle op \rangle g(\phi) \tag{2}$$

where f and g are the input values and $\langle op \rangle$ is the defined Boolean operation for this node. A test case failure is indicated when the value function for the *fail* output is 1 for at least one assignment. A test case passes if the *fail* output is 0 for all assignments.

4 Symbolic Simulation with Approximate Values

4.1 Approximation

A value function f' is an *approximation* of f , written as $f \sqsubseteq f'$, if and only if $\forall \phi. f(\phi) \sqsubseteq f'(\phi)$. Given two approximations, f' and f'' of f , f'' is said to be more approximate than f' if $f' \sqsubseteq f''$. Different approximations of a given value function are not necessarily comparable.

An *exact value* is defined as a value function which ranges over the set $\{0, 1\}$. The exact value of a node is the value function computed for that node using the Boolean operation defined for that node given that both input value functions are exact. An approximate value of a node is any value function which is an approximation of the exact value.

An approximate value for a node can be generated by simply applying the symbolic extension of the Boolean operation to the two approximate input values to produce an approximate output value. The correctness of this method is captured in the following formula:

$$f \sqsubseteq f' \wedge g \sqsubseteq g' \Rightarrow (f \langle op \rangle g) \sqsubseteq (f' \langle op \rangle g') \quad (3)$$

where $\langle op \rangle$ is the Boolean operation defined for the node, f' and g' are the input value functions, and f and g are the exact values for the input nodes. The validity of this formula is an immediate consequence of the monotonicity of AND, OR, and NOT.

Normally, input values to the simulator consist of exact values only. Values are approximated at internal nodes by the simulator according to an approximation rule. An *approximation rule* states when a value of X must be returned for some Boolean operation instead of an exact value for some set of assignments. The approximation rule limits the set of value representations allowed in order to limit the size of the internal representation of a value. Time and memory can be traded off simply by varying the approximation rule used by the simulator. This can also be done dynamically to allow the simulator to adjust the level of approximation to get the optimal trade-off between memory and time.

4.2 Improving the Approximation

Approximations are conservative, which means that the final simulation result can be approximate if some internal values are approximate. This is indicated when the value function for the *fail* output ranges over the set $\{0, X\}$. Improving the approximation consists of making values on internal nodes more exact. There are two basic ways of doing this. First, the approximation rule can be relaxed, allowing more exact values to be produced at the expense of using more memory. Second, symbolic input variables can be set to constants while using the same approximation rule internally. Since functions of fewer variables generally have smaller representations, there should be fewer approximations produced.

Our goal is to have representations that do not exceed memory capacity. Therefore, our simulator improves approximations by using a combination of

relaxed approximation rules and turning variables into constants. Our method for turning variables into constants is based on the Davis-Putnam (*DP*) algorithm [6] for proving the satisfiability of propositional formulas. The algorithm uses *case splitting* which selects one of the symbolic input variables and re-runs the simulation twice, once with the selected variable set to 0 and the other with it set to 1. *DP* recursively case splits until an exact output is generated.

Setting variables to constants is a necessary, but not sufficient condition for improving the approximation at each internal node. The minimum sufficient condition for guaranteeing that an exact value will be produced is that when all variables are set to constant values, a constant value will be output. This is guaranteed if operations on the Boolean domain produce Boolean values, which is normally the case.

The Davis-Putnam method creates a search tree. A leaf node in this tree is one in which either the *fail* output is 1 for some assignment or is the value function $\hat{0}$. If the *fail* output is 1 for some assignment, a bug is found and the test is said to be satisfiable. If the *fail* output value is $\hat{0}$, the algorithm backtracks to a previous decision and tries the other value for the variable that was split. If all branches in the case splitting tree are exhausted, then the circuit is proven to be bug-free for the property being tested and the test is said to be unsatisfiable.

The efficiency of this algorithm is determined by heuristics that select symbolic variables to case split. The goal is to split only variables that affect the outcome of the test. The variable selection heuristic works by propagating a “preferred” split variable through the circuit from the primary inputs to the final output as the circuit is simulated. The data structure for each node consists of the current value of the node and that node’s associated split variable. When the node value is updated, the node’s associated split variable is also updated. The split variable that is associated with the final *fail* output is the variable that is selected to be split.

The following algorithm guarantees that the value of a node depends on the split variable that is chosen.

1. If the value on the node is a literal, the associated variable is the literal variable.
2. If one of the inputs is a non-controlling value (e.g., 1 is non-controlling for AND) then select the other input’s associated variable.
3. Otherwise, select the associated variable of the input that has the lowest index. Normally, the user will assign lower indices to variables that are expected to be split to minimize the amount of case splitting.

4.3 Quasi-symbolic Simulation

Quasi-symbolic simulation [17] is a particular implementation of symbolic simulation with approximate internal values. Quasi-symbolic simulation restricts value functions to the set $\mathcal{Q} = \{\hat{0}, \hat{1}, \hat{X}, \hat{a}, \neg\hat{a}, \hat{b}, \neg\hat{b}, \dots\}$ where $a, b, \dots \in \mathcal{V}$. This domain is called the quasi-symbolic domain because it cannot represent all possible symbolic functions.

Quasi-symbolic value functions can be computed straightforwardly by outputting the value function \hat{X} whenever the exact output value cannot be represented using the quasi-symbolic domain. For example, the computation \hat{a} AND \hat{b} produces the value \hat{X} at the output. If the simulator case splits on a , then the value \hat{b} is produced when $a = 1$ and $\hat{0}$ when $a = 0$. Since both of these values are exact, the approximation has been improved by case splitting.

The approximation rule for quasi-symbolic simulation can be encapsulated as follows:

Approximation Rule 1 (Quasi-symbolic Rule) *If the value being produced is a function of two or more variables, output the value \hat{X} , else output the correct value from the domain \mathcal{Q} .*

Note that this rule allows all values in \mathcal{Q} to be represented by a single word in memory. Consequently, there is no possibility of memory blow up using this approximation rule.

Quasi-symbolic evaluation coupled with recursive case splitting to resolve conservativeness is surprisingly effective at performing symbolic system simulation. The small size of the quasi-symbolic domain causes don't care node values to be turned quickly into \hat{X} values. Since case splitting occurs only over symbolic variables on care inputs, there is no penalty for having many don't care symbolic variables in a test.

4.4 Quasi-symbolic Simulation with Unit Propagation

Quasi-symbolic simulation can be optimized using a procedure called *unit propagation*. Unit propagation is an implication procedure that is used in the Davis-Putnam method to reduce the size of the search tree. The unit propagation algorithm we use is Propositional Constraint Propagation (PCP) [5]. In this algorithm, two sets of literals are associated with the value at each node. These sets, called C-sets and D-sets, list literals that are conjoined and disjoined respectively with the node value. For example, if the value at a node can be approximated as $\hat{X} \wedge \hat{a} \wedge \hat{b}$ where a and b are literals, then the value of this node is represented as \hat{X} with the C-set $\{a, b\}$. D-sets are constructed similarly with disjoined literals.

C-sets and D-sets for the result of a Boolean operation are computed using set intersection and union operations over the C-sets and D-sets of the input values to a node based on the Boolean operation defined for the node. For example, for the AND operator, the C-set of the output is equal to the union of the input C-sets. The D-set of the output is equal to the intersection of the input D-sets. Logical NOT is computed by swapping the C-set and D-sets and complementing each literal in the swapped set.

A C-set that contains a variable and its complement is called *basic inconsistent*. In this case the value can be replaced by the constant value $\hat{0}$; similarly a basic inconsistent D-set is replaced by $\hat{1}$.

The case splitting algorithm is modified to allow unit propagation by first examining the C-sets and D-sets of the output, and then based on these, eliminating branches in the tree. If the output value has a non-empty D-set, the test is immediately known to be satisfiable. If the output value has a non-empty C-set, then all literals in the C-set are set to the value 1. This eliminates having to explore the complemented case for each of the C-set literals, reducing the ultimate size of the tree. Literals from C-sets that are set to 1 are said to be unit propagated. After unit propagation is done, the circuit is re-simulated and checked for unit propagation again. Case splitting occurs only if the output value is \hat{X} and both the output C-set and D-sets are empty.

C/D-set based approximations allow value functions of more than one variable. However, approximations with more than one variable are strictly limited to those that can be represented as sets of conjoined or disjoined literals. The next section discusses how to relax these restrictions to allow a richer set of approximations while still controlling the amount of memory used by value functions.

5 Approximation Using Variable Categorization

To allow more general approximations, BDDs with extensions to allow approximate values are used in the simulator. The simulator manipulates these BDDs based on a categorization of the symbolic variables. Using BDDs to represent values eliminates the need for case splitting to resolve conservativeness. However, our algorithm still uses case splitting for two reasons. First, case splitting is used as part of the variable categorization algorithm and second, if BDD overflow occurs, the algorithm reverts to case splitting to resolve conservativeness.

5.1 Variable Categories

If quasi-symbolic values only are used to approximate node values, then symbolic variables can be categorized into three types from the simulator's point of view.

- *Care variables* are those that the simulator case splits on and so the result must depend on these variables.
- *Leaf node variables* are variables the output depends on but that are not case split. There can be no more than one of these at each leaf node of the search tree since quasi-symbolic evaluation can only compute functions of a single variable exactly without case splitting.
- *Don't care variables* are symbolic variables that the output *does not* depend on for this test.

These categorizes correspond roughly to the user's view of symbolic variables. In general, the user wants the simulator to split only on control variables and does not want it to split on don't cares. Data variables fall in between; if only simple equivalence checks are required, these can be handled by leaf node variables, but if more complex data manipulations is required, data variables may need to be case split.

The variable categorizations above can be used to guide the selection of the appropriate level of approximation at a given node. Our algorithm works by adaptively changing the categorization of variables as the simulation runs. Multiple simulation runs are required to categorize enough variables such that an exact result is produced.

At any point in time, the simulator has a current variable categorization. A variable is called *marked* if the simulator has categorized it as a care variable and *unmarked* if it has not. Marked variables are always care variables while unmarked variables may be care, don't care, or leaf node variables.

5.2 Approximation Using BDDs

Value functions are represented using *ternary BDDs* (TBDDs) which we define as multi-terminal BDDs that have at least three possible terminal nodes: 0, 1, and X . Marked variables can appear without restriction in TBDDs since the goal is to make functions of marked variables as exact as possible. Unmarked variables can only appear in TBDDs in a restricted way.

To enforce these restrictions, TBDDs are computed using a modified version of the *Apply* algorithm [3]. The modified *Apply* algorithm creates the subgraphs for the *if* and *else* branches for a given node and then checks to see if the node exists in a cache called the *unique table*. If the node exists in the unique table, the node is returned. If a new node needs to be created, the approximation rules are checked first. If no approximation is required, the node is created and returned. Otherwise, the node is approximated by returning the value X .

Since we want the simulator to compute C/D-sets for node values, the approximation rule used on BDD nodes with unmarked variables is that the node must be approximated unless it is part of a C-set or D-set. This is implemented using a TBDD in which the TBDD variables are marked care variables only and TBDD terminal nodes are either the ternary constants or are pointers to C/D-sets over unmarked variables.

We call this representation a *CD-MTBDD*. Since the number of terminal nodes can be no larger than $O(2^n)$ for n CD-MTBDD variables and each C/D-set contains only a single instance of a given variable, the worst case size of a CD-MTBDD is exponential in the number of marked care variables only. C/D-sets normally stay small and thus, value functions on don't care nodes represented by CD-MTBDDs generally stay very small using this policy. At the same time, case splitting is reduced to a minimum since functions consisting only of care variables and leaf node variables will appear at the output as exact values.

5.3 Policies for Adaptive Variable Categorization

Initially, variables in the simulator are either all marked or all unmarked. During a simulation run, the simulator reclassifies variables by marking them or unmarking them as appropriate. Our policy is based on the observation that large system-level tests have many more don't care variables than care variables. Thus, our policy is to start with all variables unmarked. The split variable

that is associated with the final *fail* output after a simulation run is known to be a care variable. The simulator can mark this variable as a care variable.

Our algorithm does not immediately mark a variable as a care variable when it is discovered. Instead, variables are case split until a leaf node is discovered. If the run is satisfiable, the test stops, but if backtracking is necessary, the variable that is backtracked is marked as a care variable. The reasoning behind this is that, in our experiments we have seen that if the test is satisfiable, the first leaf node is generally satisfiable. Thus, this heuristic gets to the first leaf node as fast as possible using quasi-symbolic simulation only and if that is not satisfiable, it assumes the test is unsatisfiable and starts marking care variables to allow BDDs to be created to reduce case splitting.

5.4 BDD Overflow Handling

BDD overflow occurs when both input BDDs exist, but there is not enough memory to create the necessary output node. Overflow is handled within the framework of approximation. If the maximum node limit has been exceeded and there is no room to create a new node, our algorithm simply returns the value X instead of creating a new node. Thus, overflow handling can be characterized as simply another approximation rule.

The variable selection heuristic must be modified to allow for overflow. Normally, when the simulator has a choice of variables as the preferred split variable for some node, marked care variables are given lower priority than unmarked variables to allow a new care variable to be discovered after each run. If overflow occurs, a marked care variable must be chosen to be split and this variable must be given priority over unmarked variables when selecting a preferred split variable. Our algorithm selects the variable for the CD-MTBDD node that caused the overflow as the preferred split variable. If multiple nodes overflow, the overflow variable with the lowest index is selected.

6 Experiments

We have implemented a prototype Verilog based symbolic system simulator that supports adaptive variable categorization, case splitting to handle BDD overflow, and uses CD-MTBDDs internally. The CD-MTBDD implementation was built on top of the CMU BDD package [12]. This section reports on the results of running some typical test cases on a large representative system-level circuit.

The test design we use for these experiments is an industrial bus to network bridge for a distributed shared memory multiprocessor [16]. The properties we are verifying use only the bus portion of the design which consists of approximately 140K gates and approximately 2,402 state bits. There are two different sets of tests. In the first test, we are looking for a particular hard to find bug, and in the second test, we are trying to exercise all possible data transfers from bus to network. These tests focus on the bus to network data transfer portion of the design because this is the most complicated part of the chip. This area of the chip had the most bugs during system simulation.

6.1 Experiment 1

For the first experiment, an initial test was written and then debugged using our simulator. For each run, we recorded the results of the test, modified the test and re-ran it until the bug was discovered. Some tests were satisfiable, some were unsatisfiable, and some had timeouts due to test case bugs. Satisfiable tests indicated test case bugs or the hardware bug being discovered, and unsatisfiable tests indicated that we were searching in the wrong area for the bug. A device timeout was typically caused by an error driving a request such that the device never responded.

The initial testing was done using quasi-symbolic simulation with C/D-sets [17]. We have re-run these tests to show that the CD-MTBDD-based method improves performance without sacrificing any of the advantages of quasi-symbolic simulation on all three types of tests. There were 47 variants of the test run. Table 2 shows the results of running each of these tests in summary form based on whether the test was satisfiable due to a test case error (TESTERR) satisfiable due to a device timeout (TIMEOUT) unsatisfiable because the wrong area was being searched (UNSAT) or satisfiable due to the bug being found (BUG.) The first two columns indicate case type and how many test cases there were for each case. The columns labeled “quasi-symbolic” report the average number of evaluations (times the simulator was run to complete all case splitting) and time for each test using quasi-symbolic values only. Note, that these were run on a version of the simulator optimized for quasi-symbolic values only. The last two columns give these same values using CD-MTBDD-based approximations.

Table 2. Results of Directed Test Experiment

<i>test</i>	tests	<i>quasi-symbolic</i>		<i>CD-MTBDD-based</i>	
		<i>evals</i>	<i>time</i>	<i>evals</i>	<i>time</i>
TESTERR	17	3.8	30.8	3.0	46.6
TIME	20	1.7	50.1	1.9	94.8
UNSAT	9	52.3	445.9	7.8	131.9
BUG	1	78	863.0	17	363.6

The results show that the amount of case splitting is virtually identical for satisfiable tests between the two methods. This is because all the satisfiable tests stopped at the first leaf node and, thus, no BDDs were created due to our variable marking policy that does not mark variables until a leaf node is hit. The minor difference in the average amount of case splitting between the two methods is due to changing the order of control variables to always come before data variables in the BDD-based tests. This ordering of control variables before data variables was not a requirement in the original algorithm that used quasi-symbolic values only. The difference in execution times between the two

methods for the satisfiable cases is due to inefficiencies in the off-the-shelf BDD package we were using.

The unsatisfiable cases show that CD-MTBDDs reduce the amount of case splitting by a factor of 6.7 on average with a maximum of 13.9 and a minimum of one for one case that only required a single evaluation even with quasi-symbolic values. The maximum number of marked care variables over all the unsatisfiable tests was five. There was no BDD overflow for any of these cases and the largest BDD created was 17 nodes despite the fact that the number of don't care variables ranged from 590 to 1697 amongst all the unsatisfiable tests. This low size is due to the variable categorization policy and the unmarked variable BDD restrictions that convert values quickly to quasi-symbolic values in don't care nodes.

6.2 Experiment 2

In this experiment, a general data transfer test was written in which symbolic control variables select all possible combinations of events that affect data transfer. This test was expected to be unsatisfiable since there were no known bugs in the area being tested. This experiment started with all symbolic control variables set to a constant value in the test. We then performed a number of runs in which each run increased by one the number of control variables that were made symbolic. The graph in figure 1 plots execution time versus the number of control variables that were made symbolic in each test for both the quasi-symbolic only case and BDD-based approximation case with the quasi-symbolic cases being run on the optimized quasi-symbolic simulator.

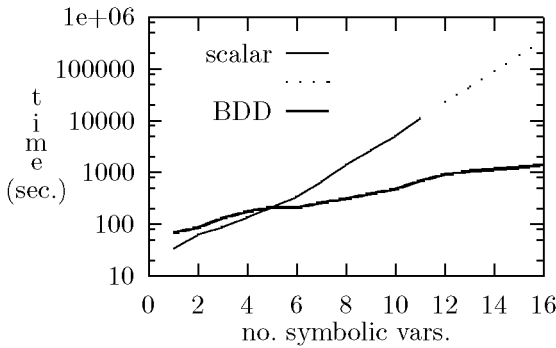


Fig. 1. Execution Time of Quasi-Symbolic and CD-MTBDD Based Approximations

This figure shows that execution time increases exponentially using quasi-symbolic values only. Using CD-MTBDDs, the growth is still exponential, but at a much lower rate. Quasi-symbolic tests were aborted due to excessive case

splitting when more than 11 symbolic care variables were present. At this point, the execution time was roughly doubling for each additional control variable that was made symbolic as is indicated by the dashed line in the plot. To understand the difference between quasi-symbolic and CD-MTBDD-based simulation further, we need to look at how various other parameters scale.

Table 3 lists the values of various parameters for each run. The first column lists the number of variables that were case split using quasi-symbolic only simulation and this is equal to the number of control variables made symbolic in a given run. This also turns out to be the number of variables that were marked as care variables during simulation with CD-MTBDDs. The second column gives the number of simulator runs required to complete the test using quasi-symbolic values only; the number of evaluations roughly doubles for each added symbolic variable. The next column gives the number of case splits required when using CD-MTBDDs. The relationship here is that the number of case splits is roughly double the number of marked care variables. This is due to our policy of not marking a variable as a care variable until a leaf node is hit and then marking variables one by one during backtracking. Thus, for each variable marked, there are two evaluations, one going down the tree and one going back up. The cases for which the number of CD-MTBDD splits is less than double the number of marked variables are due to unit propagation. Consequently, the amount of case splitting has been reduced by an exponential factor using CD-MTBDDs without a substantial increase in the time per evaluation.

The next two columns give BDD package statistics. The first of these columns indicates the largest BDD that was created in each test and the last is the total number of BDD nodes created. The largest BDD in each case is remarkably small considering that these tests created over 300 BDD variables including control, data, and don't care variables. These small sizes are due to the unmarked variable restrictions. It is hard to gauge this effect exactly since we cannot easily determine which nodes are don't cares and which are not. The total number of BDD nodes created grows exponentially, but these are mostly BDDs of size ten nodes or less.

7 Conclusion

Symbolic system simulation has the potential to be better than directed and random testing in verifying large system-level designs with mixed control and data logic. Straightforward BDD-based symbolic simulation is not optimized for system-level testing in which there are many don't care inputs. This paper presented an algorithm that uses approximations on internal nodes and heuristics based on variable categorization that allow the simulator to automatically select the appropriate level of abstraction at each node. The key to making this work is having restrictions on the BDDs that represent values on nodes. If one of these restrictions is violated, it indicates that the node is either a don't care node or if it is a care node, that further case splitting must occur. In either case, the amount of case splitting is not affected by making this node more approximate.

Table 3. Results of Experiment 2

<i>control vars</i>	<i>quasi-symbolic case splits</i>	<i>BDD case splits</i>	<i>max BDD size</i>	<i>tot. BDD nodes</i>
1	3	3	1	2352
2	6	4	3	4435
3	8	6	3	4478
4	12	8	3	4739
5	20	10	3	5422
6	32	10	3	6309
7	62	12	7	13001
8	124	14	14	38775
9	246	16	22	110729
10	450	16	45	259236
11	993	17	82	570216
12	-	18	123	867923
13	-	20	123	897016
14	-	22	123	936028
15	-	24	123	1000766
16	-	26	151	1121177

We also presented a method for handling BDD overflow that uses the built-in case splitting mechanism to control BDD size at the expense of increased simulation run time. Our experiments show that CD-MTBDD-based approximations improve performance compared to using quasi-symbolic approximations without increasing the probability of memory overflow significantly. We have not demonstrated that the set of abstraction policies we chose is optimal and in the future, we hope to explore different policy tradeoffs.

References

1. V. Bertacco, M. Damiani, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Proc. of 36th Design Automation Conf.*, pages 391–396, 1999.
2. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. of 36th Design Automation Conf.*, pages 317–320, 1999.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. H. Cho, G. Hachtel, E. Macii, B. Pleisser, and F. Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Trans. on Comp.-Aided Design of Integrated Circuits and Systems*, 15(12):1465–1478, December 1996.
5. M. Dalal. Efficient propositional constraint propagation. In *Proc. of the Tenth National Conf. on Artificial Intelligence (AAAI-92)*, pages 409–414, 1992.

6. M. Davis, G. Logemann, and D. Loveland. Machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
7. M. Ganai, A. Aziz, and A. Kuehlman. Augmenting simulation with symbolic algorithms. In *Proc. of 36th Design Automation Conf.*, pages 385–390, 1999.
8. S. Govindaraju, D. L. Dill, A. J. Hu, and M. A. Horowitz. Approximate reachability with bdds using overlapping projections. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.
9. R. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In *1996 IEEE International Conference on Computer-Aided Design*, pages 146–151, 1996.
10. P. Jain and G. Gopalakrishnan. Efficient symbolic simulation based verification using the parametric form of boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1005–1015, 1994.
11. R. Jones, M. Aagard, and C.-J. Seger. Formal verification using parametric representations of boolean constraints. In *Proc. of 36th Design Automation Conf.*, pages 402–407, 1999.
12. D. E. Long. Cmu bdd package, 1993.
13. D. Moundanos, J. A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, January 1998.
14. K. Ravi, K. McMillan, T. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc. of 35th Design Automation Conf.*, pages 445–450, 1998.
15. C.-J. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.
16. W.-D. Weber et al. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proc. of the 24th Annual Intl. Symp. on Computer Architecture (ISCA97)*, 1997.
17. C. Wilson and D. L. Dill. Reliable verification using symbolic simulation with scalar values. In *Proceedings of the 37th Design Automation Conference*, June 2000. Los Angeles, CA.

A Theory of Consistency for Modular Synchronous Systems[★]

Randal E. Bryant¹, Pankaj Chauhan¹, Edmund M. Clarke¹, and Amit Goel²

¹ Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213, USA

{bryant,pchauhan,emc}@cs.cmu.edu

² Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, PA 15213, USA
{agoel}@ece.cmu.edu

Abstract. We propose a model for modular synchronous systems with combinational dependencies and define consistency using this model. We then show how to derive this model from a modular specification where individual modules are specified as Kripke Structures and give an algorithm to check the system for consistency. We have implemented this algorithm symbolically using BDDs in a tool, SpecCheck. We have used this tool to check an example bus protocol derived from an industrial specification. The counterexamples obtained for this protocol highlight the need for consistency checking.

1 Introduction

The correctness of a system is defined in terms of its specification. In model checking [6], for example, a model of the design is verified against a set of temporal logic properties. However, specifications might have errors themselves. The authors have discovered errors in protocol specifications for the PCI bus and the CoreConnect bus [4,9]. This highlights the need to examine specifications more carefully.

These problems often occur due to the limitations of natural languages used to describe specifications. However, even formal specifications can have problems. Sometimes, it is not possible to realize the specification in any implementation, while at other times, the system could deadlock. Bus specifications often allow *combinational dependencies* which are sometimes desirable for efficiency reasons. These dependencies might cause a module to exhibit deadlock only on certain inputs. Problems like these are often attributed to inconsistencies in the

□□[★] This work was supported in part by the National Science Foundation under Grant no. CCR-9803774 and in part by the MARCO/DARPA Gigascale Silicon Research Center (<http://www.gigascale.org>). Their support is gratefully acknowledged. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

specification. We propose a theory that formalizes the notion of *consistency* for modular synchronous systems.

The main contributions of this paper are a model for representing modular synchronous systems that incorporates combinational dependencies, a definition of consistency for this model, a method to obtain this model from any given specification in which each module can be described by a Kripke structure [6], and an algorithm to check for consistency for this model.

We motivate the paper with small examples in which the specifications are expressed in Linear Temporal Logic (LTL) [6]. They demonstrate inconsistent behavior as explained.

Example 1. Consider the following trivial example. The specification for a module consists of two properties:

- a. $\mathbf{G} \neg reset$: *reset* should never be asserted.
- b. $reset$: *reset* should be asserted initially.

We can see that it is impossible to satisfy this specification.

Example 2. As another example, consider an arbiter that receives requests from two masters with the following specification:

- a. $\mathbf{G}(req_0 \rightarrow \mathbf{X} ack_0)$: If master 0 makes a request, it should be acknowledged in the next cycle.
- b. $\mathbf{G}(req_1 \rightarrow \mathbf{X} ack_1)$: If master 1 makes a request, it should be acknowledged in the next cycle.
- c. $\mathbf{G} \neg (ack_0 \wedge ack_1)$: Both masters should not be acknowledged at the same time.

If both req_0 and req_1 are asserted, there is no valid next state assignment for ack_0 and ack_1 . Whatever the arbiter does, at least one property will **not** be satisfied.

Before the next example, let us consider a *Master* and *Slave* that communicate using the signals *req* and *ack*. Suppose that one of the requirements of the Master is expressed by the LTL formula $\mathbf{G}(req \rightarrow req \mathbf{U} ack)$. Once asserted, *req* should stay asserted until it is acknowledged. This property allows for *req* to be deasserted in the same cycle in which *ack* is asserted. Therefore, the next state value of *req*, i.e. req' , depends not only on the current state value of *req* and *ack*, but also on ack' , the *next state value* of *ack*. The timing diagram in figure 1 illustrates this. A combinational dependency such as this can sometimes lead to inconsistent behavior as seen in the next example.

Example 3. The following is a specification for a device with output *req* and inputs *busy* and *ack*:

- a. $\mathbf{G}(req \rightarrow req \mathbf{U} ack)$: Once request is asserted it remains asserted till the request is acknowledged.
- b. $\mathbf{G}(busy \rightarrow \mathbf{X} \neg req)$: A request should not be made if the bus is busy.

In this case, the problem arises when a request has already been made and the bus becomes busy (possibly because of some other device). The second property

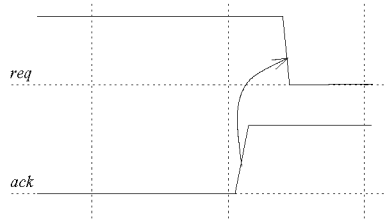


Fig. 1. Timing diagram showing combinational dependency between *req* and *ack*.

requires *req* to be deasserted but the first property then requires that *ack* be asserted. However, *ack* is an input and cannot be controlled by the device under consideration. This means that the specification breaks down on certain inputs. This problem is similar to that of receptiveness defined by Dill in [8] for *asynchronous systems*. However, if the specification of the module controlling *ack* is constrained ($\mathbf{G}(\text{busy} \rightarrow \mathbf{X} \text{ack})$), then the problem disappears.

In our model, which we call the Synchronous Consistency Model (SCM), we divide a synchronous step into several phases which we call *micro-transitions*. After each micro-transition, the state of the system is partially updated. A transition to the next state is complete after the last micro-transition. The partial updates to the states due to micro-transitions are represented by *micro-states*. Micro-transitions allow us to capture combinational dependencies between signals.

The concept of micro-transitions is not entirely new. Most hardware description languages such as VHDL and VERILOG have some notion of combinational dependency. ESTEREL [2] allows dynamically scheduled sub-rounds. Alur and Henzinger [1] also break a round (synchronous step) into sub-rounds (micro-transitions) in the Reactive Module Language(RML). While Alur and Henzinger describe an operational modeling language, we do not present any language. Instead, we describe how to derive the SCM from modular specifications and check it for consistency. In RML, the model explicitly specifies the behavior of sub-rounds. We want the micro-transitions to be synthesized automatically from high level, declarative specifications.

For our experiments, we have used LTL for specification and tableau construction to derive Kripke structures. Other approaches might be more suitable. Shimizu et. al. [12] describe a monitor based formal specification methodology for modular synchronous systems. Clarke et. al. [7] describe a way to obtain executable protocol specification and describe algorithms that enable them to debug specifications. However, neither of them incorporate combinational dependencies. We believe that these two approaches are complementary to our work. Given monitors or executable models, our approach can be used to check for consistency.

The organization of the rest of the paper is as follows. In Section 2, we discuss preliminaries needed for the rest of the paper. We define the Synchronous Consistency Model and **consistency** for systems using this model in Section 3.

Section 4 describes how to derive a Synchronous Consistency Model from a modular specification. We present our algorithm to check the consistency of an SCM in Section 5. In section 6 we introduce SpecCheck, a prototype tool we have built for consistency checking. We also describe a system and its specification, based on an industrial bus protocol, that we checked using our tool. Finally, we conclude our paper with directions for future research in section 7. In Appendix A we present the complete specification for the example system described in Section 6.

2 Preliminaries

As mentioned in the introduction, we derive a Synchronous Consistency Model from modular specifications where each module is specified as a Kripke structure. Section 2.1 defines a Kripke structure. Section 2.2 defines parallel composition for Kripke structures. We will use parallel composition to derive a global state transition graph for the specification. The example specifications we use in this paper are expressed in LTL, which is defined in Section 2.3.

2.1 Kripke Structures

A Kripke structure [6] T is a tuple (S, S_0, R, AP, L) where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation, AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

If a specification is expressed in temporal logic, then tableau construction methods [5,10] produce a Kripke structure for that specification. For example, the method described for LTL properties in [5] produces a Kripke structure with every path that satisfies the LTL property. Figure 2(a) shows a tableau for a module with the only requirement being $\mathbf{G}(a \rightarrow \mathbf{X} \neg b)$. Figure 2(b) shows a tableau for $\mathbf{G}(b \rightarrow b \mathbf{U} c)$.

2.2 Synchronous Parallel Composition

Let $T' = (S', S'_0, AP', L', R')$ and $T'' = (S'', S''_0, AP'', L'', R'')$ be two tableaux. The synchronous parallel composition [6] of T' and T'' denoted by $T' \parallel T''$ is the structure $T = (S, S_0, AP, L, R)$ defined as follows.

1. $S = \{(s', s'') \mid L'(s') \cap AP'' = L''(s'') \cap AP'\}$.
2. $S_0 = (S'_0 \times S''_0) \cap S$.
3. $AP = AP' \cup AP''$.
4. $L((s', s'')) = L'(s') \cup L''(s'')$.
5. $R((s', s''), (t', t''))$ if and only if $R'(s', t')$ and $R''(s'', t'')$.

If $s = (s', s'') \in S$ then we say that s' and s'' are components of s . Figure 3 shows the structure obtained from the synchronous parallel composition of the two structures in Figure 2.

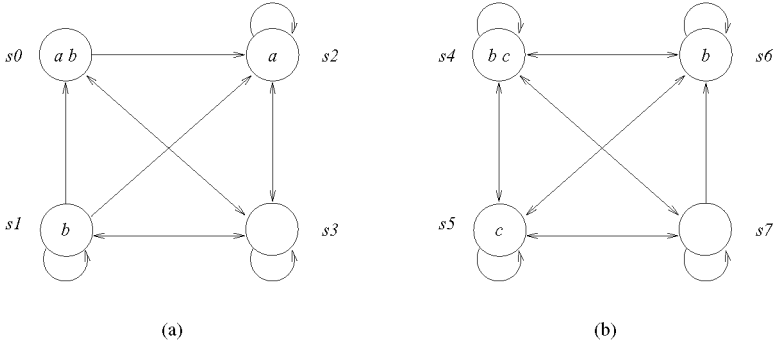


Fig. 2. (a) A tableau for $\mathbf{G}(a \rightarrow \mathbf{X} \neg b)$. (b) A tableau for $\mathbf{G}(b \rightarrow b \mathbf{U} c)$.

This definition of composition models *synchronous* behavior. States of the composition are pairs of component states that agree on the common atomic propositions. Each transition of the composition involves a joint transition of the two components. The parallel composition of more than two tableaux is defined similarly.

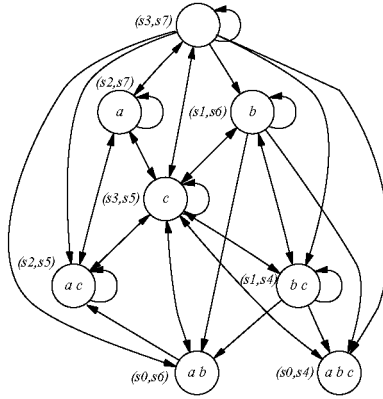


Fig. 3. Synchronous Parallel Composition of the Tableaux in Figure 2

2.3 LTL

For a set of atomic proposition AP , the set of LTL formulas is defined as follows:

- an atomic proposition $p \in AP$ is an LTL formula.
- If f and g are LTL formulas, then $\neg f$, $f \vee g$, $\mathbf{X} f$ and $f \mathbf{U} g$ are LTL formulas.

The following abbreviations are also used:

- $f \wedge g = \neg(\neg f \vee \neg g)$
- $\mathbf{F} f = \text{true} \mathbf{U} f$
- $\mathbf{G} f = \neg \mathbf{F} \neg f$

For a complete discussion of LTL and its semantics refer to [6].

3 Synchronous Consistency Model

The model we are proposing augments the state transition graph of the system with extra information to capture combinational dependencies. As discussed in the introduction, these dependencies are an important aspect of many bus protocol specifications [9].

Definition 1 (Synchronous Consistency Model (SCM)). A synchronous consistency model M is a six-tuple (S, R, P, C, n, Φ) where

1. S is the set of observable states.
2. $R \subseteq S \times S$ is a transition relation.
3. P is the set of micro-states.
4. $C : S \rightarrow P$ maps observable states to micro-states.
5. $n \in \mathbf{N}$ is the number of micro-transitions in a sequential step.
6. $\Phi = \{\phi_{s,i} \subseteq P \times P \mid s \in S, 1 \leq i \leq n\}$ is the set of micro-transition relations.

(S, R) defines a synchronous state transition graph. We say that the states in S are *observable*. The micro-states in P capture the intermediate stages of computation within a synchronous transition and are *unobservable*. For each observable state $s \in S$, there is a corresponding micro-state $C(s) \in P$. We represent a synchronous transition from $s \in S$ as a sequence of micro-transitions beginning at $C(s)$. Without loss of generality, we assume that all transitions between observable states have the same number n of micro-transitions. The relation $\phi_{s,i} \subseteq P \times P$ describes the allowed partial updates in the i^{th} micro-step, starting from the observable state s . We require that $\phi_{s,i}$ be defined for all observable states s and for each micro-step up to n . In order to define what *consistency* means, we need the following definitions.

A *valid sequence* of micro-states is required to begin in a micro-state corresponding to an observable state and obey the micro-transition relations beginning in that state.

Definition 2 (Valid Sequence). A sequence of micro-states $\langle p_0, p_1, \dots, p_l \rangle$ is a valid sequence with respect to a model $M = \langle S, R, P, C, n, \Phi \rangle$ iff there exists an observable state $s \in S$ such that $C(s) = p_0$, and for all i such that $1 \leq i \leq l$, $(p_{i-1}, p_i) \in \Phi_{s,i}$.

A *valid transition sequence* is a valid sequence that ends in a micro-state corresponding to an observable state. A valid transition sequence corresponds to a synchronous transition in the state transition graph (S, R) .

Definition 3 (Valid Transition Sequence). A sequence $\langle p_0, p_1, \dots, p_n \rangle$ is a valid transition sequence with respect to a model $M = \langle S, R, P, C, n, \Phi \rangle$ iff there exist observable states s and s' such that:

- 1 $C(s) = p_0$
- 2 for all i such that $1 \leq i \leq n$, $(p_{i-1}, p_i) \in \Phi_{s,i}$
- 3 $C(s') = p_n$
- 4 $(s, s') \in R$

A valid sequence which can not be extended to form a valid transition sequence represents inconsistent behavior. We call these sequences *divergent sequences*.

Definition 4 (Divergent Sequence). A sequence is a divergent sequence with respect to a model $M = \langle S, R, P, C, n, \Phi \rangle$ iff it is a valid sequence with respect to M , and it is not a prefix of any valid transition sequence in M .

Definition 5 (Micro-transition Conformance). The set of micro-transition relations Φ for a model $M = \langle S, R, P, C, n, \Phi \rangle$ conforms to the transition relation R iff for every $(s, s') \in R$ there exists a valid transition sequence beginning and ending in micro-states $C(s)$ and $C(s')$, respectively.

We are now ready to define what it means for a model to be *consistent*. If a model satisfies this definition, then the consistency problems described in the introduction can be avoided.

Definition 6 (Consistent Model). A model $M = \langle S, R, P, C, n, \Phi \rangle$ is said to be consistent iff it satisfies the following conditions:

- 1 $S \neq \emptyset$.
- 2 R is total, i.e., for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
- 3 The set of micro-transition relations Φ conforms to the transition relation R .
- 4 There are no divergent sequences in M .

If the model is derived from a specification, then the first condition ensures that the specification is satisfiable, while the second checks for the absence of deadlocks. Conformance implies that the micro-transition relation implements the global transition relation. The absence of divergent sequences guarantees that the system does not get stuck after a valid sequence of micro-transitions.

4 Deriving an SCM from Modular Specifications

We have defined *consistency* for an SCM. We now describe a way to derive an SCM $M = (S, R, P, C, n, \Phi)$ from modular specifications. We are given a set of modules M_i and a partial order \prec on AP . The relation \prec is used to capture combinational dependencies. If b depends on a then $a \prec b$. Each module is described by a Kripke structure T_i .

The variables occurring in the structure for each module are classified either as input or output variables for that module. We say that a module M_i controls

its output variables and require that a variable be controlled by exactly one¹ module. This allows us to define a function γ from the set of variables to the set of modules. $\gamma(v_a) = M_a$ iff the variable v_a is controlled by the module M_a .

Let $T = (S, S_0, AP, L, R)$ be the global Kripke structure obtained by the parallel composition $T_1 \parallel T_2 \parallel \dots \parallel T_m$. We restrict the states of T to only those reachable from S_0 to obtain (S, R) .

The set of micro-states P is defined as the power set of the atomic propositions in T , i.e., $P = 2^{AP}$. The mapping C from observable states to micro-states is then defined by L .

We use \prec to derive the number of micro-transitions in a synchronous transition, n , and the set of micro-transition relations, Φ . We can partition the atomic propositions AP into disjoint sets by levelizing \prec . A set in the leveled partition may contain variables controlled by different modules. We further split each set to obtain the partition Y so that the resulting sets Y_i in $Y \doteq \langle Y_1, \dots, Y_n \rangle$ have variables controlled by only one module. The number of sets n in Y determines the number of micro-transitions in M . Y satisfies the following properties:

1. $\bigcup_{i=1}^n Y_i = AP$, i.e. the partitioning is exhaustive,
2. $Y_i \cap Y_j = \emptyset, i \neq j$, i.e. the sets in the partition are disjoint,
3. $[x_i \prec x_j \text{ and } x_i \in Y_k, x_j \in Y_l] \Rightarrow k < l$, i.e. the set of atomic propositions is *levelized* by \prec ,
4. $\forall v_a, v_b \in Y_i \cdot \gamma(v_a) = \gamma(v_b)$, i.e. all variables in the same partition are controlled by the same module.

Define the function $\pi : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, m\}$ such that $M_{\pi(k)}$ is the controlling module for variables in Y_k , i.e., $\gamma(v) = M_{\pi(k)}$ for all $v \in Y_k$. This function is well defined since the controlling module for all variables in Y_k is the same.

The partitioning Y tells us the order in which next state variables are updated. We begin with an observable micro-state p_0 in which no variable has been updated yet. Then the next state values of variables in Y_1, Y_2, \dots, Y_n are computed in that order. The i^{th} micro-transition depends on transitions in the tableau for the controlling module for that micro-step.

Consider the example in Figure 4, which shows an observable state s such that $L(s) = \{a, b\}$ in the global Kripke structure of specifications for two modules M_1 and M_2 . M_1 controls a and b and its properties are $\mathbf{G}(a \rightarrow \mathbf{X} \neg b)$ and $\mathbf{G}(b \rightarrow b \mathbf{U} c)$ (Figure 3). M_2 controls c and is unconstrained. The partitioning $Y = \{\{a\}, \{c\}, \{b\}\}$ of variables is a legal partitioning for the given partial order $c \prec b$. The micro-state corresponding to s is p_0 . Since there are three sets in Y , there are three micro-transitions. Figure 4, shows all the legal micro-transitions for the system starting from observable state s .

In the second micro-transition, M_2 updates the next state value of c , while the next state value of a has already been updated, and the next state value of b will be updated in the next micro-transition. Let s_j be the component of s in

¹ This allows only closed systems. However, that is not a problem since we can always introduce an extra module to control the primary inputs to the system, without restricting their behavior.

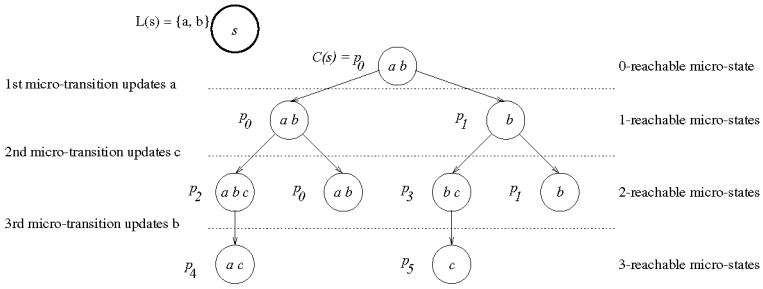


Fig. 4. Legal micro-transitions possible from $C(s_0) = p_0 = \{a, b\}$ for specifications $b \rightarrow b \cup c$ and $a \rightarrow \mathbf{X} \neg b$

M_2 . Suppose that after the first micro-transition we get to p_1 , then the partially updated value of a is false. This restricts transitions in M_2 to those next states where a is false. M_2 can update c to either true or false. If c is updated to false, then M_1 is restricted to those transitions in which the next state values of a and c are both false. But there is no such transition in M_1 from s_j . Hence there is no third micro-transition from p_1 starting from s . The following recursive definitions formally capture this idea.

Definition 7 (i-Reachability). A micro-state $p_k \in P$ is i -reachable from $s \in S$ iff there exists a valid sequence of length $i + 1$ beginning with $C(s)$ and ending with p_k .

Definition 8 ($\Phi_{s,i}$). For $p_a, p_b \in P$, $(p_a, p_b) \in \Phi_{s,i}$ iff p_a is $(i - 1)$ -reachable from s and there exist $s_j, s_k \in S_{\pi(i)}$ such that:

1. s_j is a component of s .
2. $R_{\pi(i)}(s_j, s_k)$.
3. $p_b = (p_a - Y_i) \cup (L_{\pi(i)}(s_k) \cap Y_i)$. The micro-state p_b is obtained from p_a by retaining the values of all variables except those in Y_i . The updated value of Y_i is extracted from s_k .
4. $p_a \cap (Y_1 \cup \dots \cup Y_{i-1}) \cap AP_{\pi(i)} = L_{\pi(i)}(s_k) \cap (Y_1 \cup \dots \cup Y_{i-1})$. The values of variables updated before the i^{th} micro-transition agree with their values in s_k .

For every observable state s there is only one 0-reachable micro-state, $C(s)$. This allows us to compute $\Phi_{s,1}$ which in turn gives us the set of 1-reachable micro-states from s . Using these definitions recursively, we can compute Φ . The following theorem guarantees that the Φ obtained in this manner conforms to R .

Theorem 1. The set of micro-transition relations Φ obtained from definitions 7,8 for the SCM $M = (S, R, P, C, n, \Phi)$ conforms to the transition relation R .

Proof: Given $s = (s_1, s_2, \dots, s_m)$ and $s' = (s'_1, s'_2, \dots, s'_m)$ such that $R(s, s')$, we need to prove that there exists a valid transition sequence beginning and ending in $C(s)$ and $C(s')$ respectively. We construct a sequence $\langle p_0, p_1, \dots, p_n \rangle$ as follows:

- $p_0 = C(s)$
- $p_i = (p_{i-1} - Y_i) \cup (L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i)$ for all $1 \leq i \leq n$

If we can prove that $(p_{i-1}, p_i) \in \Phi_{s,i}$ for all $i \in \{1, \dots, n\}$ and that $C(s') = p_n$ then $\langle p_0, p_1, \dots, p_n \rangle$ is one such valid transition sequence. To see that $(p_{i-1}, p_i) \in \Phi_{(s,i)}$, we consider $s_{\pi(i)}$ and $s'_{\pi(i)}$. State $s_{\pi(i)}$ is a component of s . $R(s, s') \Rightarrow R_{\pi(i)}(s_{\pi(i)}, s'_{\pi(i)})$, and $p_i = (p_{i-1} - Y_i) \cup (L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i)$ by construction. It then remains to be shown that $p_{i-1} \cap (Y_1 \cup \dots \cup Y_{i-1}) \cap AP_{\pi(i)} = L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \dots \cup Y_{i-1})$.

We now prove by induction that $p_i \cap (Y_1 \cup \dots \cup Y_i) = L(s') \cap (Y_1 \cup \dots \cup Y_i)$. It then follows that (a) $p_{i-1} \cap (Y_1 \cup \dots \cup Y_{i-1}) \cap AP_{\pi(i)} = L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \dots \cup Y_i)$, and (b) $p_n \cap (Y_1 \cup \dots \cup Y_n) = L(s') \cap (Y_1 \cup \dots \cup Y_n)$. Since the partitioning Y is exhaustive, $(Y_1 \cup \dots \cup Y_n) = AP$, $p_n = L(s') = C(s')$.

For the base case, we consider p_1 . By our construction $p_1 = (p_0 - Y_1) \cup (L_{\pi(1)}(s'_{\pi(1)}) \cap Y_1)$. This implies $p_1 \cap Y_1 = (L_{\pi(1)}(s'_{\pi(1)}) \cap Y_1)$. Since $s'_{\pi(1)}$ is a component of s' , $p_1 \cap Y_1 = L(s') \cap Y_1$.

For our inductive hypothesis, assume that $p_{i-1} \cap (Y_1 \cup \dots \cup Y_{i-1}) = L(s') \cap (Y_1 \cup \dots \cup Y_{i-1})$.

$$\begin{aligned}
 p_i &= [p_{i-1} - Y_i] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
 \Rightarrow p_i \cap (Y_1 \cup \dots \cup Y_i) &= [(p_{i-1} - Y_i) \cap (Y_1 \cup \dots \cup Y_i)] \\
 &\quad \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap (Y_1 \cup \dots \cup Y_i)] \\
 &= [p_{i-1} \cap (Y_1 \cup \dots \cup Y_{i-1})] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
 &= [L(s') \cap (Y_1 \cup \dots \cup Y_{i-1})] \cup [L_{\pi(i)}(s'_{\pi(i)}) \cap Y_i] \\
 &\quad \text{by induction hypothesis} \\
 &= [L(s') \cap (Y_1 \cup \dots \cup Y_{i-1})] \cup [L(s') \cap Y_i] \\
 &\quad \text{because } s'_{\pi(i)} \text{ is a component of } s' \\
 \Rightarrow p_i \cap (Y_1 \cup \dots \cup Y_i) &= L(s') \cap (Y_1 \cup \dots \cup Y_i) \square
 \end{aligned}$$

5 Algorithm to Check Consistency

The heart of our algorithm for checking consistency is a procedure for computing the set of i -reachable micro-states from an observable state s . We compute the i -reachable micro-states from s by updating the next state values of Y_i variables in the $(i-1)$ -reachable micro-states from s . The valuation of $\tilde{Y} = Y_{i+1} \cup \dots \cup Y_n$ remains unchanged. As in the previous section, $M_{\pi(i)}$ is the controlling module for the variables in the partition Y_i . Given γ , it is easy to compute $\pi(i)$. In this computation, if we find that there exists an $(i-1)$ -reachable micro-state for

which there is no successor micro-state, then we have a divergent sequence of length $i - 1$. This implies that the model is inconsistent by definition 6.

The explicit-state algorithm **RECCHECK** shown in Figure 5 returns false if there is a divergent sequence beginning with $C(s)$. The loop in line 4 iteratively computes P_i , the set of i -reachable micro-states from s . Line 6 assigns $Y_{i+1} \cup \dots \cup Y_n$ to Y_{after} . In line 7, we initialize P_i to the empty set. The loop beginning in line 8 checks for every $(i - 1)$ -reachable micro-state p_{i-1} , if we can extend any valid sequence of length i ending in p_{i-1} . If it can be extended, then all i -reachable micro-states p_i which extend these sequences are added to P_i (line 13) and the flag *extended* is set to true. If there is a divergent sequence, *extended* remains false in line 15 and the procedure terminates by returning false. In line 9, l denotes the valuation of the variables that have been updated in the previous steps ($Y_{before} = Y_1 \cup \dots \cup Y_{i-1}$). Line 10 initializes the extended flag to false. The loop beginning in line 11 iterates over all states s_k in the image of s_j in the controlling module for the i^{th} micro-transition. Line 12 checks if s_k is compatible with the updates made in the previous micro-transitions (l). Line 17 adds Y_i to Y_{before} .

The explicit-state procedure **SPECHECK** for checking consistency is also described in Figure 5. Note that we do not check the micro-transition relation for conformance since that is guaranteed by Theorem 1. In fact, we do not explicitly construct Φ . We check for divergent sequences by the i -reachability procedure in Figure 5.

As mentioned, the algorithms presented in this section are explicit-state. We have implemented symbolic state versions of these algorithms in our tool, **SpecCheck**, described in the next section.

6 Implementation and Experimental Results

We implemented a prototype tool, **SpecCheck**, that performs consistency checking on a given specification for a synchronous modular system. The input to the tool is a list of modules along with a set of properties expressed in LTL for each module, and a partial order, \prec , on the atomic propositions used. To derive tableaux for LTL properties, we used a slight modification of the construction described by Clarke et. al. in [5] symbolically using BDDs [3]. After deriving a tableau for each LTL property, we compose tableaux of all LTL properties to derive a Kripke structure for a module and in turn compose these modules to form a global Kripke structure. There is no unique tableau for an LTL property. We have observed spurious deadlocks in some cases, depending on the tableau construction used. Our modification to the method of [5] gets rid of the spurious deadlocks in our examples. We compute the set of i -reachable micro-states, P_i , symbolically using BDDs. P_i is a set of tuples (s, p) , where $(s, p) \in P_i$ if and only if p is i -reachable from s . The set of states is encoded using $AP \cup A$, where A is a set of auxiliary state variables which are used to differentiate between states with the same labelling. The label of a state can be obtained by existentially quantifying the auxiliary variables. The set of micro-states is encoded using AP . **SpecCheck** is implemented in **Moscow ML** [15] which is an implementation of


```

    REC_CHECK( $s, n, Y, \pi, \langle T_1, T_2, \dots, T_m \rangle$ )
1   $Y_{\text{before}} \leftarrow \emptyset$ 
2   $Y_{\text{after}} \leftarrow Y$ 
3   $P_0 \leftarrow L(s)$ 
4  for  $i = 1$  to  $n$ 
    {
5       $s_j \leftarrow \text{COMPONENT}(s, \pi(i))$ 
6       $Y_{\text{after}} \leftarrow Y_{\text{after}} - Y_i$ 
7       $P_i = \emptyset$ 
8      for every  $p_{i-1} \in P_{i-1}$ 
        {
9           $l \leftarrow p_{i-1} \cap Y_{\text{before}}$ 
10          $\text{extended} \leftarrow \text{false}$ 
11         for every  $s_k \in S_{\pi(i)}$ 
            {
12             if  $(R_{\pi(i)}(s_j, s_k) \text{ and } (L_{\pi(i)}(s_k) \cap Y_{\text{before}}) = l)$ 
13                  $P_i = P_i \cup \{(p_{i-1} - Y_i) \cup (L_{\pi(i)}(s_k) \cap Y_i)\}$ 
14                  $\text{extended} \leftarrow \text{true}$ 
            }
15         if  $(\text{extended} = \text{false})$ 
16             return false
        }
17      $Y_{\text{before}} \leftarrow Y_{\text{before}} \cup Y_i$ 
    }
18 return true

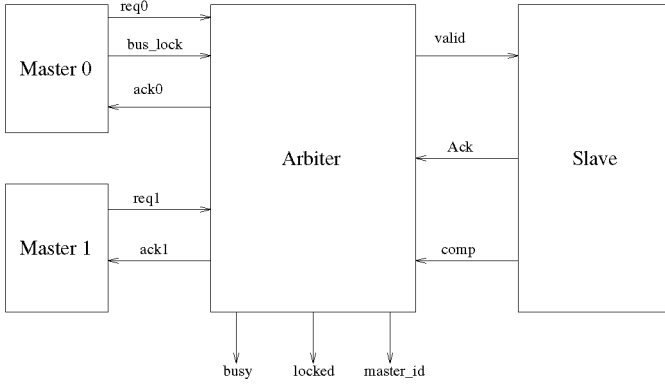
```

```

    SPEC_CHECK( $\langle T_1, T_2, \dots, T_m \rangle, \gamma, \prec$ )
1   $(S, R) \leftarrow \text{COMPOSE}(\langle T_1, T_2, \dots, T_m \rangle)$ 
2  if  $(S = \emptyset)$  return false
3  if  $R$  is not total return false
4   $\dot{Y} = \text{LEVELIZE}(AP, \prec)$  /* levelize  $\prec$  */
5   $(Y, \pi) = \text{MODPARTITION}(\dot{Y}, \gamma)$ 
   /* refine  $\dot{Y}$  so that each partition has variables controlled by only one module */
6  for every  $s \in S$ 
    {
7      if  $(\text{REC\_CHECK}(s, n, Y, \pi, \langle T_1, T_2, \dots, T_m \rangle) = \text{false})$  return false
    }
8  return true

```

Fig. 5. REC_CHECK, algorithm to check for the absence of divergent sequences and SPEC_CHECK, algorithm to check consistency

**Fig. 6.** Example System

standard ML [11]. The symbolic computations are performed using MuDDy [13] which is an ML interface to the BuDDy [14] BDD package.

We used our tool to check specifications for the system depicted in figure 6. Two version of specifications for this system are described in more detail in the appendix A. The system consists of two master devices and one slave connected to an arbiter.

The general flow of control is as follows. The masters request control of the bus using their respective *req* signals. The arbiter passes the request to the slave (*valid*) and when the slave acknowledges the request (*Ack*), the arbiter passes the acknowledgement to the requesting master (*ack0* or *ack1*). After acknowledging a request, the slave indicates completion of the request by asserting *comp*.

The *busy* signal is used to indicate when the bus is busy. Master 0 can also try to lock the bus by asserting *bus_lock* along with its request. If the bus is locked, this is indicated by *locked*. In the locked state, the arbiter ignores requests made by master 1.

The *master_id* signal indicates which master currently controls the bus. If the bus is idle, the value is undefined. Arbitration occurs when the bus is free. The first master to make a request when the bus is free is granted the bus (by setting *master_id* appropriately). If both masters make a request, then the request from master 0 gets priority. In version A, the arbiter asserts *valid* one cycle after arbitration, whereas in version B, *valid* is asserted in the same cycle. In the appendix A, we have precisely described the properties of these modules in LTL.

Figure 7(a) shows a counterexample for version A, demonstrating a deadlock, while figure 7(b) demonstrates a receptiveness problem with version B.

In figure 7(a), there is no valid next state because of the following properties.

1. $\mathbf{G}(((req0 \vee req1) \wedge \neg busy) \rightarrow \mathbf{X} valid)$: If there is a request and the bus not busy, then in the next cycle *valid* must be asserted.
2. $\mathbf{G}(locked \wedge \neg req0 \rightarrow \mathbf{X} \neg valid)$: If the bus is locked and master 0 is not making a request, then *valid* cannot be asserted in the next cycle.

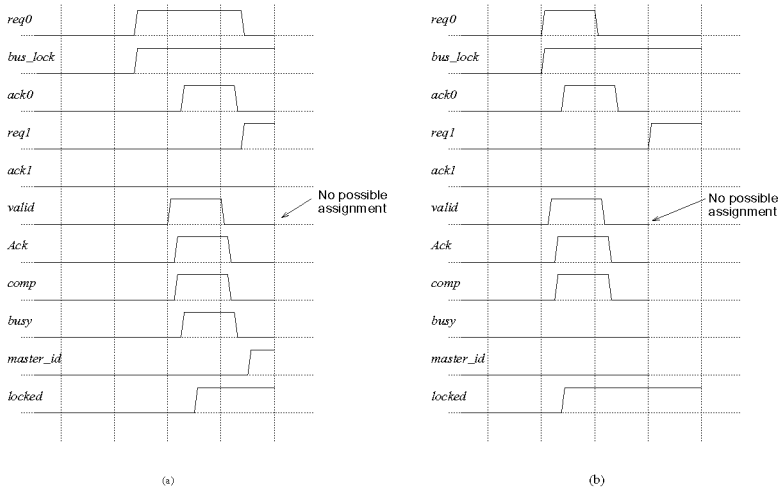


Fig. 7. (a) Deadlock in the specifications. (b) Receptiveness problem in specifications

Property 1 requires *valid* to be asserted in the next state, while property 2 requires it to be deasserted. Hence there is no possible next state in the trace shown. This inconsistency can be removed by replacing the first property with the following two properties:

- 1(a). $\mathbf{G}((req0 \wedge \neg busy) \rightarrow \mathbf{X} valid)$: If master 0 makes a request and the bus is not busy, then in the next cycle *valid* must be asserted.
- 1(b). $\mathbf{G}((req1 \wedge \neg (busy \vee locked)) \rightarrow \mathbf{X} valid)$: If master 1 makes a request and the bus is neither busy nor locked, then in the next cycle *valid* must be asserted.

In figure 7(b), an incomplete trace for version B is shown. In the last cycle, only *req0*, *bus_lock*, *req1* and *locked* are assigned. If *req1* had remained deasserted in the last cycle, there would have been no problem. But because it is asserted, there is no possible assignment for *valid*. The following properties lead to this receptiveness problem.

3. $\mathbf{G}((\neg valid \wedge \neg busy) \rightarrow \mathbf{X}((req0 \vee req1) \leftrightarrow valid))$: If *valid* is deasserted and the bus is not busy, *valid* is asserted in the next cycle if and only if there is a request in that cycle.
4. $\mathbf{G}(locked \wedge valid \rightarrow req0)$: When the bus has been locked by master 0, ignore all requests by master 1, so *valid* just reflects master 0's requests.
5. $\mathbf{G}(locked \rightarrow \mathbf{X}(locked \leftrightarrow bus_lock))$: If the bus is locked, it remains locked while master 0 keeps *bus_lock* asserted.

Just like for version A, this inconsistency can be resolved by replacing property 3 with the following two properties:

- 3(a). $\mathbf{G}((\neg \text{valid} \wedge \neg \text{busy} \wedge \neg \text{locked}) \rightarrow \mathbf{X}(\text{req0} \leftrightarrow \text{valid}))$: If *valid* is deasserted and the bus is not busy but locked, *valid* is asserted in the next cycle if and only if master 0 makes a request in that cycle.
- 3(b). $\mathbf{G}((\neg \text{valid} \wedge \neg \text{busy} \wedge \neg \text{locked}) \rightarrow \mathbf{X}((\text{req0} \vee \text{req1}) \leftrightarrow \text{valid}))$: If *valid* is deasserted and the bus is neither busy nor locked, *valid* is asserted in the next cycle if and only if there is a request in that cycle.

7 Conclusion and Future Research

We have proposed a theory for consistency of modular synchronous systems with combinational dependencies, and developed an algorithm that allows us to check for inconsistencies. The algorithm has been implemented in a prototype tool, SpecCheck, which has been used to find bugs in the examples described in the paper.

Nevertheless, there are a number of directions for future research. We have assumed that the partial order \prec on AP is specified by the user. We believe that it may be possible to derive this order automatically from a specification in temporal logic. The starting point in our check for consistency is a modular specification given as a collection of Kripke structures. We have observed that it is possible to obtain spurious deadlocks with SpecCheck, depending on the tableau construction method used. Similar issues are discussed in [6]. A better understanding of tableau construction methods and how they are used in SpecCheck should enable us to eliminate the spurious deadlocks that we have observed. We need to understand better what consistency means for actual implementations. The standard notion of simulation between an implementation and a more abstract model is not sufficient to guarantee consistency for the implementation. Finally, we need to extend SpecCheck to handle hierarchical systems. We believe that this can be done within our current framework.

References

1. R. Alur and T.A. Henzinger. "Reactive modules." In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pp. 207-218. 1996.
2. G. Berry, G. Gonthier. "The synchronous programming language ESTEREL: Design, semantics, implementation." Technical Report 842, INRIA. 1988.
3. R. E. Bryant. "Graph-based algorithms for boolean function manipulation." *IEEE Transactions on Computers*, C-35(8), pp. 677-691. 1986.
4. P. Chauhan, E. Clarke, Y. Lu, D. Wang. "Verifying IP-Core based System-On-Chip designs." In *Proceedings of the IEEE ASIC/SOC Conference*, pp. 27-31. 1999.
5. E. Clarke, O. Grumberg, H. Hamaguchi. "Another look at LTL model checking." *Formal Methods in System Design*, 10, pp. 47-71. 1997.
6. E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press. 1999.
7. E. Clarke, Y. Lu, H. Veith, D. Wang, S. German. "Executable Protocol Specification in ESL." *Formal Methods in Computer-Aided Design (FMCAD'00)*. 2000.
8. D.L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

9. A. Goel, W. R. Lee. "Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core." *37th ACM/IEEE Design Automation Conference*. 2000.
10. D. E. Long. "Model Checking, Abstraction and Compositional Reasoning." PhD Thesis, Carnegie Mellon University, 1993.
11. Milner, Tofte, Harper, MacQueen. *The Definition of Standard ML*. MIT Press, 1997.
12. K. Shimizu, D. L. Dill, A. J. Hu. "Monitor-Based Formal Specification." *Formal Methods in Computer Aided Design (FMCAD'00)*. 2000.
13. K. F. Larsen, J. Lichtenberg. MuDDy. Version 1.7.
<http://www.itu.dk/research/muddy>.
14. J. L. Nielsen. BuDDy—A Binary Decision Diagram Package. Version 1.7.
<http://www.itu.dk/research/buddy>.
15. S. Romanenko, P. Sestoft. Moscow ML. Version 1.44.
<http://www.dina.dk/~sestoft/mosml.html>.

A Example Specifications

Slave Specification

Slave specifications are identical for both versions. We require that:

1. Slave should acknowledge only valid requests.
2. Slave should not assert *comp* before the first request is acknowledged.
3. *comp* should be asserted only once per request. If *comp* has been asserted once, then it will be deasserted until another request has been acknowledged.

Master Specification

Version A

1. Once asserted *req0* should remain asserted until it is acknowledged.
2. Once asserted *req1* should remain asserted until it is acknowledged.
3. Master 0 can request to lock the bus by asserting *bus_lock* only when *req0* is asserted.

Version B

1. Once asserted *req0* can only be deasserted one cycle after the cycle in which *ack0* is asserted.
2. Once asserted *req1* can only be deasserted one cycle after the cycle in which *ack1* is asserted.
3. Master 0 can request to lock the bus by asserting *bus_lock* only when *req0* is asserted.

Valid Signal

Version A

1. If there is a request and the bus is not busy, then in the next cycle *valid* must be asserted.
2. Once asserted *valid* can only be deasserted one cycle after the cycle in which *Ack* is asserted.
3. If *valid* is deasserted, and either the bus is busy or there is no request then *valid* should remain deasserted in the next cycle.
4. If *Ack* is asserted, *valid* should be deasserted in the next cycle.
5. If the bus is locked and master 0 is not making a request, then *valid* cannot be asserted in the next cycle.

Version B

1. If *valid* is deasserted and the bus is not busy, *valid* is asserted in the next cycle if and only if there is a request in that cycle.
2. Once asserted *valid* can only be deasserted one cycle after the cycle in which *Ack* is asserted.
3. If *valid* is deasserted and the bus is busy then *valid* should remain deasserted in the next cycle.
4. If *Ack* is asserted in response to a *valid* then *valid* should be deasserted in the next cycle.
5. When the bus has been locked by master 0, ignore all requests by master 1.

Arbiter Acknowledgement Signals

Version A

1. If current master is 0 and an acknowledge is received in the next cycle then *ack0* is asserted in the next cycle, otherwise *ack0* remains deasserted.
2. If current master is 1 and an acknowledge is received in the next cycle then *ack1* is asserted in the next cycle, otherwise *ack1* remains deasserted.

Version B

1. If current master is 0 and an acknowledge is received then *ack0* is asserted, otherwise *ack0* remains deasserted.
2. If current master is 1 and an acknowledge is received then *ack1* is asserted, otherwise *ack1* remains deasserted.

Busy Signal

Version A

1. If *busy* is deasserted, it remains deasserted until an *Ack* is received.
2. If an *Ack* is received, *busy* is asserted.
3. Once asserted *busy* can only be deasserted one cycle after the cycle in which *comp* is asserted.
4. If *comp* is asserted, *busy* should be deasserted in the next cycle.

Version A	Version B
Slave Specification	
$\mathbf{G}(Ack \rightarrow valid)$ $\neg comp \mathbf{U} Ack$ $\mathbf{G}(comp \rightarrow \mathbf{X}(\neg comp \mathbf{U} Ack))$	
Master Specification	
$\mathbf{G}(req0 \rightarrow req0 \mathbf{U} ack0)$ $\mathbf{G}(req1 \rightarrow req1 \mathbf{U} ack1)$ $\mathbf{G}(\neg bus_lock \rightarrow (\neg bus_lock \mathbf{U} req0))$	$\mathbf{G}(req0 \wedge \neg ack0) \rightarrow \mathbf{X} req0$ $\mathbf{G}(req1 \wedge \neg ack1) \rightarrow \mathbf{X} req1$ $\mathbf{G}(\neg bus_lock \rightarrow (\neg bus_lock \mathbf{U} req0))$
Valid Signal	
$\mathbf{G}(((req0 \vee req1) \wedge \neg busy) \rightarrow \mathbf{X} valid)$ $\mathbf{G}((valid \wedge \neg Ack) \rightarrow \mathbf{X} valid)$ $\mathbf{G}(\neg valid \wedge \neg((req0 \vee req1) \wedge \neg busy) \rightarrow \mathbf{X} \neg valid)$ $\mathbf{G}(Ack \rightarrow \mathbf{X} \neg valid)$ $\mathbf{G}(locked \wedge \neg req0 \rightarrow \mathbf{X} \neg valid)$	$\mathbf{G}((\neg valid \wedge \neg busy) \rightarrow \mathbf{X}((req0 \vee req1) \leftrightarrow valid))$ $\mathbf{G}((valid \wedge \neg Ack) \rightarrow \mathbf{X} valid)$ $\mathbf{G}(\neg valid \wedge busy \rightarrow \mathbf{X} \neg valid)$ $\mathbf{G}(Ack \wedge valid \rightarrow \mathbf{X} \neg valid)$ $\mathbf{G}(locked \wedge \neg req0 \rightarrow \neg valid)$
Arbiter Acknowledgement Signals	
$\mathbf{G}(\neg master_id \wedge \mathbf{X} Ack \leftrightarrow \mathbf{X} ack0)$ $\mathbf{G}(master_id \wedge \mathbf{X} Ack \leftrightarrow \mathbf{X} ack1)$	$\mathbf{G}(ack0 \leftrightarrow Ack \wedge \neg master_id)$ $\mathbf{G}(ack1 \leftrightarrow Ack \wedge master_id)$
Busy Signal	
$\mathbf{G}(\neg busy \rightarrow \neg busy \mathbf{U} Ack)$ $\mathbf{G}(Ack \rightarrow busy)$ $\mathbf{G}(busy \wedge \neg comp \rightarrow \mathbf{X} busy)$ $\mathbf{G}(comp \rightarrow \mathbf{X} \neg busy)$	$\mathbf{G}(\neg busy \rightarrow \mathbf{X}(busy \leftrightarrow (Ack \wedge \neg comp)))$ $\mathbf{G}(busy \rightarrow \mathbf{X}(\neg busy \leftrightarrow comp))$
Master_id Signal	
$\mathbf{G}((locked \vee busy) \rightarrow (master_id \leftrightarrow \mathbf{X} master_id))$ $\mathbf{G}(\neg(locked \vee busy) \rightarrow \mathbf{X}(req0 \rightarrow \neg master_id))$ $\mathbf{G}(\neg(locked \vee busy) \rightarrow \mathbf{X}(req1 \wedge \neg req0 \rightarrow master_id))$	
Locked Signal	
$\mathbf{G}(\neg locked \rightarrow \mathbf{X}(locked \leftrightarrow bus_lock \wedge (\neg master_id \wedge Ack)))$ $\mathbf{G}(locked \rightarrow \mathbf{X}(locked \leftrightarrow bus_lock))$	

Table 1. LTL specifications for the example system depicted in Figure 6*Version B*

1. If the bus is not busy then in the next cycle the bus is busy if and only if the slave acknowledge a request and does not complete it in that cycle.
2. If the bus is busy then the bus remains busy till the slave asserts *comp*. Once *comp* is asserted, the bus is no longer busy.

Master_id Signal

The arbiter arbitrates by setting *master_id* to denote which master is controlling the bus. Both versions behave identically when setting *master_id*.

1. *master_id* remains unchanged if the bus is either busy or locked.
2. If the bus is neither busy nor locked then master 0 is given control of the bus if it makes a request.
3. If the bus is neither busy nor locked then master 1 is given control of the bus if it makes a request and master 0 is not making a request.

Locked Signal

1. If the bus is not locked then in the next cycle the bus is locked if and only if a bus locking request from master 0 is acknowledged by the slave.
2. If the bus is locked, it remains locked while master 0 keeps *bus_lock* asserted.

The initial state of all the signals is deasserted.

Verifying Transaction Ordering Properties in Unbounded Bus Networks through Combined Deductive/Algorithmic Methods^{*}

Michael Jones and Ganesh Gopalakrishnan

University of Utah, 50 S. Central Campus Dr. Rm. 3190
Salt Lake City, UT 84112-9205
{mjones,ganesh}@cs.utah.edu

Abstract. Previously [MHG98,MHJG00], we reported our efforts to verify the producer/consumer transaction ordering property for the PCI 2.1 protocol extended with local master IDs. Although our efforts were met with some success, we were unable to show that all execution traces of *all* acyclic PCI networks satisfy this transaction ordering property. In this paper, we present a verification technique based on network symmetry classes along with a manually derived abstraction that allows us to show, at the bus/bridge level, that all execution traces of all acyclic PCI networks satisfy the transaction ordering property. This now completed case study (modulo the validity of the axioms used to characterize the abstraction) suggests several avenues for further work in combining model-checking (algorithmic methods) and theorem-proving (deductive methods) in judicious ways to solve infinite-state verification problems at the bus/interconnect level. It is a concrete illustration of partitioning concerns where designers can specify bus protocols in an operational semantics (rule-based) style, invent abstractions, and carry out finite-state model-checking while verification experts can establish formal properties of the abstraction.

1 Introduction

Verifying transaction ordering properties in unbounded bus networks is of growing importance, given the prevalence of busses and systems on chip (SOC) interconnects. In this paper we present an approach that combines deductive and algorithmic methods to achieve exhaustive verification of safety properties over unbounded branching networks. An unbounded branching network is a class of network topologies in which there is no limit on the number of nodes that may be included in the network so long as the resulting network is both acyclic and completely connected. Our work is characterized by several features:

- It is a concrete illustration of partitioning concerns where designers can specify bus protocols in an operational (rule-based) style, invent abstractions, and carry out finite-state model-checking while verification experts can establish formal properties of the abstraction.

^{*} Supported in part by NSF Grant No. CCR-9800928

- It is a concrete case study involving the popular PCI 2.1 bus and the emerging Virtual Components Interface (VCI) SOC bus
- It is a concrete demonstration of how combined algorithmic/deductive methods may be deployed in practice.
- It provides a method to reason about non-trivial protocols over unbounded branching networks.

The subject of the case study was the PCI 2.1 protocol. The PCI 2.1 local bus protocol is an open standard that implements a message passing system over acyclic bus/bridge networks. The PCI standard supports two message types and allows message reordering and deletion in certain situations. Despite the possibility of reordering and deletion, the PCI protocol is intended to obey the producer/consumer (PC) transaction ordering property. Unfortunately, the published PCI standard violates the PC property due to a phenomenon called completion stealing [Cor96]. Completion stealing allows the consumer to read from the data address before the producer has written the new data value. In this case study, we verify a corrected version of the PCI protocol, which has been extended with local master IDs. Local master IDs were proposed as a solution to the completion stealing problem. For the purposes of this report, *PCI* will refer to the PCI protocol extended with local master IDs and *PCI_e* will refer to the original, erroneous, PCI protocol without local master IDs. We do not address cycle or physical level details of the PCI specification in this report.

Our overall approach to solving this verification problem was to use a manually derived, but formally justified, symmetry reduction on branching networks. This symmetry reduction reduced instances of the PC property over arbitrary PCI networks to one of four network symmetry classes¹. Unfortunately, each of these reduced networks has an infinite number of states. A further modification of the PCI protocol was required to reduce the unbounded number of states in each reduced network, resulting in a protocol called *PCI'*. We then created a PVS proof in which we showed that the PCI protocol operating on bus/bridge networks is a trace inclusion refinement of the *PCI'* protocol operating on the networks from the symmetry classes. We then checked all execution traces of *PCI'* in all four reduced networks in just under 2 minutes of execution time using the Mur ϕ model checker. No violations were found in model checking. The refinement proof taken together with the model checking results allow us to conclude that all traces of all PCI networks satisfy the PC property.

After exhaustively verifying the PC property for *PCI*, we turned our attention to two related verification problems: verifying the PC for the original *PCI_e* protocol without local master IDs and verifying an aspect of *PCI/VCI* interaction. We modified the *PCI'* protocol to create a *PCI'_e* protocol such that *PCI_e* is a trace inclusion refinement of *PCI'_e*. Using the same symmetry reduction as in the *PCI'* verification, we found a violation of the PC property in the *PCI'_e* protocol which could be traced back to completion stealing in the *PCI_e* protocol. We also verified that the PCI protocol satisfies an in-order transmission policy

¹ A symmetry class is a kind of equivalence class in which the members of a class are identical under a well-defined set of transformations.

required for transmitting packets chains in the Virtual Sockets Interface Alliance (VSIA, see [All00]) VCI on-chip bus standard. This behavior is important should the on-chip bus VCI and the PCI be connected in an SOC environment.

The key contributions of this paper are:

- The exhaustive verification of the PC property for all execution traces in all PCI networks at the bus/bridge level.
- An abstraction technique for reducing unbounded sets of branching networks to finite sets of reduced networks. This is done using a symmetry reduction that ignores bridge boundaries and is parameterized by the number of agents in the property being verified.
- An abstraction technique for handling the unbounded states in each reduced network created by the symmetry reduction in the previous contribution. The fact that each reduced network has an unbounded number of states arises from a mechanism used to acknowledge delayed transactions in the PCI protocol. The unbounded states are reduced based on an invariant identified using a PVS model of PCI.
- An illustration that a combination of model checking and theorem proving together with network symmetry reductions and rule-based notation is an effective approach for reasoning about non-trivial protocols over branching networks

The importance of the PCI case study itself stems from PCI being an important open I/O standard which finds wide use in computer systems; and is slated for migration into next generation technologies—such as systems on chips. This work extends our previous PCI verification work by formalizing the refinement proof. The refinement proof allows us to state the transaction ordering verification problem as a model checking problem. We plan to develop a network protocol verification tool based on the techniques used in this case study.

In the next section we sketch pertinent details of the PCI protocol and PC transaction ordering property with the aim of presenting the specific verification problem addressed in this report. Section 3 contains an overview of our particular solution with a more detailed description of each solution step given in each of the subsections. In section 4 we give our verification results for PCI_e without local master IDs and the VSIA packet chain ordering problem. Section 5 contains related work and section 6 summarizes this our observations relative to this case study.

2 Problem Definition

In this section, we sketch an overview of the PCI protocol and network topology and describe the PC transaction ordering property specified. A PCI network is an acyclic hyper-graph of agents and bridges connected by busses such that there exists a unique path between any two agents. Agents are connected to one local bus while bus bridges connect (“bridge”) two local busses. An example of a PCI

network appears in the left half of Figure 3. As shown in the figure, agents and bridges each contain two queues; one in each direction. The *opposite queue* of the top queue in a bridge is the bottom queue (pointing in the opposite direction) in the same bridge, and vice versa.

A PCI network supports two types of transactions: *posted* and *delayed*. Posted transactions are *unacknowledged* transactions that can be neither deleted nor bypassed. Delayed transactions are acknowledged transactions that can be bypassed at any time by other delayed or posted transactions. A *committed* delayed transaction is a delayed transaction that has been attempted, but not necessarily latched, on the local bus. Delayed transactions may be dropped unless they have been committed. Delayed transactions leave a trail of committed copies of themselves in every bridge through which they pass. The response to a delayed transaction is called its *completion*. Completions travel back from target to source following the trail of copies of the matching delayed transaction. Completions can be dropped and bypassed—except for delayed write completions, which can not be passed. A posted transaction is considered complete at a bridge or agent as soon as it has been issued on the local bus. Delayed transactions are considered complete at a bridge or agent only when the completion has returned to that bridge or agent. The rules for bypassing and dropping transactions are intended to prevent deadlock while preserving the transaction ordering required by the PC property.

The PCI specification requires that PCI bus/bridge networks have the PC transaction ordering property. For the purposes of PCI, the PC property is stated as follows.

If the following preconditions are satisfied:

- (1) An agent, the producer (*Prod*), issues two write transactions: W_{Data} (a posted or delayed write transaction to the address *Data*) followed by W_{Flag} (a posted or delayed write transaction to the address *Flag*),
- (2) An agent, the consumer *Cons* issues two Delayed Read Request transactions R_{Flag} followed by R_{Data} ,
- (3) W_{Flag} is issued on the originating bus after the completion of W_{Data} on the originating bus,
- (4) R_{Data} is committed on the originating bus after the completion of R_{Flag} on the originating bus,
- (5) R_{Flag} is completed on the destination bus after W_{Flag} completes on the destination bus.

Then, assuming no other agents write to the data address, the value returned by *Cons* R_{data} is the value written by *Prod* W_{data} .

The central problem in this case study then is to show that all execution traces of all PCI networks satisfy the PC property.

3 Solution

Our solution to the problem of exhaustively verifying the PC transaction ordering property is outlined in figure 1. The solution can be divided into three steps

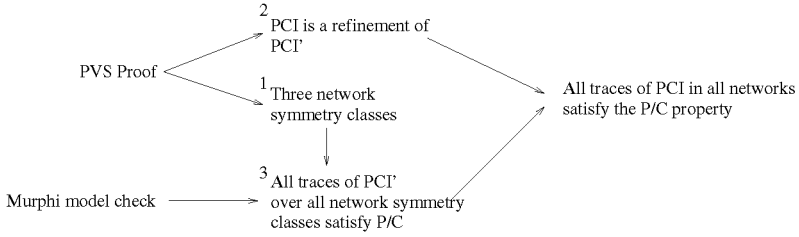


Fig. 1. Outline of PCI transaction ordering verification.

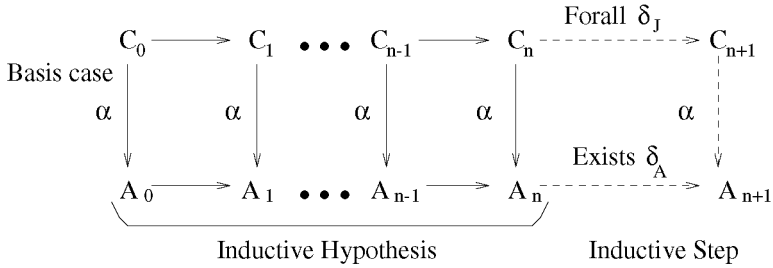


Fig. 2. Outline of proof that PCI is a refinement of PCI'.

and each step is labeled 1,2 or 3 in the figure. First, we use a manually discovered branching network symmetry reduction on PCI bus/bridge networks which requires us to transform the PCI protocol into PCI' (see section 3.1). Second, we use an interactive higher-order logic theorem prover to prove that the PCI protocol on PCI networks is a trace inclusion refinement of the PCI' protocol on the reduced networks (section 3.2). Third, we use an explicit state model checker to show that all execution traces on the reduced networks satisfy the PC property (section 3.3).

The goal of the branching network symmetry reduction is to map arbitrary instances of the PC property over a PCI network to a finite number of instances over a finite number of reduced networks. In [MHG98] we showed that we could map all instances of the PC property on PCI to one of four reduced networks. This structural symmetry reduction allows us to cover of all instances of the PC property over all PCI networks by checking the execution traces of four abstract networks. The symmetry reduction applies to all unbounded branching networks, not just PCI networks. Using the symmetry reduction requires us to modify the the PCI protocol so that each reduced network has a finite number of states; resulting in the PCI' protocol.

Our goal in showing that PCI is a trace inclusion refinement of PCI' was to formally establish that safety properties of PCI' hold for PCI as well. This allows us to apply model checking results from PCI' to PCI. Proving trace inclusion refinement requires showing that for every concrete trace, there exists an abstract

trace such that each state in the abstract trace is equal to the abstraction of the corresponding concrete state [AL91]. Trace inclusion is conservative with respect to safety properties. This means that properties violated by PCI' are not necessarily violated by PCI . In the context of PCI , for each concrete trace, σ , we need to show that there exists an abstract trace, σ_A in PCI' , such that the following relationship holds:

$$\begin{aligned} \forall \sigma \in \text{PCI}. \{ \sigma = (A_0, C_0), (A_1, C_1), (A_2, C_2) \dots (A_n, C_n) \\ \Rightarrow \exists \sigma_A \in \text{PCI}'. \sigma_A = A_0, A_1, A_2 \dots A_n \} \end{aligned}$$

in which for every A_i in σ_A $A_i = \text{abs}(C_i)$ for some abstraction function abs . The proof is outlined in figure 2. As shown in the figure, the relationship will be shown by induction on the length of σ . The operational semantics of each protocol were defined inductively using a set of inference rules. A set of inference rules, one per PCI transition, describe the operational semantics of each PCI and PCI' transition in terms of the transition's effect on the system state. The set of reachable states is inductively defined, starting from an initial state, as the parallel composition of the inference rules. An inference rule can be applied to a reachable state to create a new reachable state if and only if the rule's preconditions are satisfied by the current reachable state. A state is reachable if and only if it can be derived from the initial state by a sequence of applications of the inference rules. All of the rules needed to define the operational semantics of PCI and PCI' can be found at [Jon00]. The inference rules facilitate the inductive refinement proof by suggesting a natural way to perform a case split on the construction of the next state. The rules were then used to split the inductive step by showing that the abstraction of every application of every PCI rule, labeled δ_J , has a corresponding application of a PCI' rule, labeled δ_A .

Our goal in model checking PCI' was to show that every trace in every reduced network satisfies the PC property. The inference rules describing the operational semantics of PCI' were used to define a model of PCI' for use in the Mur ϕ model checker [ID96]. We checked all PCI' traces of all four reduced networks in just under 2 minutes. No violations of the PC property were found in PCI' which, due to the refinement relation, means that all traces of the PCI protocol on all PCI networks also do not violate the PC property.

The next three sections are each devoted to explaining the network symmetry reduction, refinement proof and model checking results in more detail.

3.1 Network Symmetry Reduction

The symmetry reduction allows us to map instances of the PC property to one of four reduced networks, but requires us to modify the PCI protocol to keep the number of states in each reduced network finite. Consider the problem of showing that n agents in an unbounded branching network satisfy a certain property. In the context of PCI there may also be arbitrarily many other agents which might affect the property being verified; in addition, the paths between any two agents may contain arbitrarily many bridges and busses. Despite the fact that each

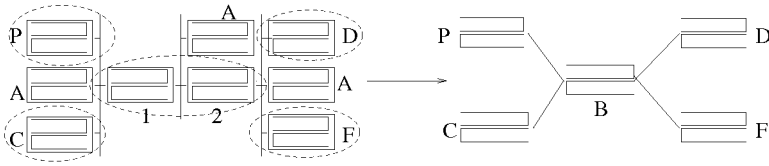


Fig. 3. Example of a network symmetry reduction

network has a (large) finite number of states, there are an unbounded number of networks. One way to reduce this problem over an unbounded number of networks to a bounded number of networks is to consider the symmetry classes induced by ignoring agents and paths incidental to the property being verified and to ignore the lengths of the paths between agents central to the property being verified.

Figure 3 contains an example of such a symmetry reduction on an instance of the PC on a PCI network. The network on the left side of the figure contains an instance of the PC property defined on a network with seven agents and two bridges. The producer, consumer, data and flag agents are labeled P, C, D and F while the other agents are all labeled with A's. The parts of the network on the left side of the figure that contribute to the reduced network have been circled. The reduced network appears on the right side of the figure. In the reduced network, all agents incidental to the PC property, labeled with A's, are ignored. In addition, the path between bridges 1 and 2 in the center of the network on the left have been coalesced into path B in the network on the right. The transactions in the queues of bridges 1 and 2 are concatenated and placed in path B. One can add arbitrarily many busses, bridges and agents to the network on the left and it will still reduce to the same network on the right.

Despite the fact that we are now concerned with only four reduced networks, the PCI protocol over each of these reduced networks has an unbounded number of states. Recall that the contents of each bridge in a path are concatenated to form the contents of a path. Although each bridge contains a bounded number of transactions, there may be arbitrarily many bridges in a path. This means there may be arbitrarily many transactions in a path. We can reduce the number of transactions in a path by ignoring transactions other than those required by the property being verified. However, in the case of PCI, this still allows paths with an unbounded number of transactions. Recall that as delayed transactions travel between source and destination, they leave a trail of committed copies of themselves in each bridge through which they pass. Each of these committed copies, one per bridge, are then concatenated and placed in the corresponding path in the reduced network—again leading to an unbounded number of states in the reduced network. We eliminate this source of unboundedness by keeping only the newest copy of a committed delayed transaction in the state of the reduced network. This reduction is justified by a PCI invariant and required several modifications to the PCI' protocol. The modifications to the PCI' protocol were

required so that the upcoming PVS proof that PCI is a refinement of PCI' would go through.

3.2 PVS Refinement Proof

The refinement proof was carried out in the PVS theorem prover using three theories which described PCI, PCI' and the abstraction. We first describe each of the three theories then discuss the PVS refinement proof. All three theories and the complete refinement proof can be found at the following URL [Jon00].

PCI Theory. The theory describing the concrete PCI protocol contained ten inference rules and several supporting definitions. Each of the ten rules describe a PCI transition and were adapted from [CSZ97, PCI95]. A set of 10 inference rules which describe the operational semantics of PCI can also be found at [Jon00].

Given a reachable state, a rule constructs a new reachable state if the preconditions are met. The preconditions to each state involve at most a bridge (or agent) and its adjacent bridge (or agent). If the preconditions are met, the next reachable state is constructed by modifying the contents of a bridge (or agent) and its adjacent bridge (or agent). The limited scope of each rule allows us to apply the same set of inductive rules to networks with different topologies. This is because each rule depends only on the connection between two entities, rather than the topology of the entire network.

The PCI theory contains 551 lines of PVS, including comments. The theory is built on top of a theory of unbounded finite sequences which are used to model the queues. Two rules cover the movement of posted transactions while the remaining eight cover the movement of delayed transactions and their completions. Starting from a network in its initial state, the entire set of reachable states for the network can be computed by repeatedly applying rules whenever and wherever their preconditions are satisfied.

PCI' Theory. The PCI' rules were designed so that they would mimic every effect of the PCI rules on the reduced networks while preserving as many properties of PCI as possible. If the PCI' rules mimic every effect of the PCI rules then we can show that PCI is a refinement of PCI'. Suppose we chose the PCI' rules to be the CHAOS rule-set, in which all actions are always allowed. In this case, we can certainly show that PCI is a refinement of CHAOS, but we can not show that CHAOS has any useful properties. By carefully designing the PCI' protocol, we create a protocol which is refined by PCI but which also has useful properties. A set of inference rules describing the operational semantics of the PCI' protocol can also be found at [Jon00].

The theory describing the PCI' protocol contained fifteen rules and several supporting definitions. The PCI' theory contains more rules than the PCI theory because additional rules are needed to compensate for the network information lost in the abstraction. The additional rules in the PCI' theory stem from the loss of queue boundaries and loss of committed copies of delayed transactions under the abstraction. Figure 4 illustrates the ambiguity resulting from the loss of queue boundaries. Both network fragments on the left, $N1$ and $N2$, contain a single transaction, $T1$ and $T2$. A path boundary is indicated by the dashed

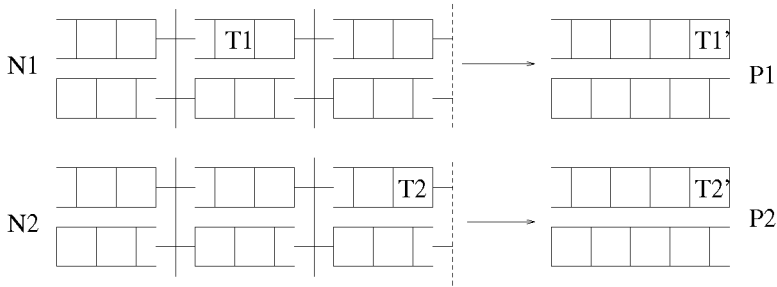


Fig. 4. Example of ambiguity in reduced network paths.

line. Despite the fact that $T2$ has reached the final position in the path, since no transactions appear ahead of $T1$ in $N1$: both $T1$ and $T2$ map to the head of the paths $P1$ and $P2$ under the abstraction. This would not be a problem, except the new transactions created by latching $T1$ and $T2$ in $N1$ and $N2$ appear in different locations in $P1$ and $P2$. In $P1$, the new transaction would appear in front of $T1'$ and in $P2$ the new transaction would appear at tail of the next path after $P2$. As a result, PCI' requires two versions of certain latching rules to cover both cases. Similar instances of ambiguity involving suppressed committed copies of delayed transactions necessitated additional rules in the PCI' model.

The PCI' theory is 507 lines long, including comments. The PCI' theory uses the same theory of finite sequences to represent paths. Of the fifteen rules, three cover the movement of posted transactions and the other twelve cover the movement of delayed transactions and their completions.

Abstraction Theory. The abstraction theory describes the abstraction function that relates states in PCI to states in PCI' . The abstraction function performs the network symmetry reduction and eliminates transactions (as discussed in section 3.1) so that each reduced network has a finite number of states. Rather than define the abstraction using a set of definitions and prove lemmas about the abstraction for use in the refinement proof, we chose to state needed properties about the abstraction as axioms and use the axioms directly in our refinement proof. We chose an axiomatic rather than a definitional theory to save the time required to prove each of the abstraction lemmas and instead focus on the refinement proof. A total of 51 axioms about the abstraction were used in the refinement proof. The axioms describe the effects of the network symmetry reduction, predicates which can be inferred about a PCI' state from a PCI state and the effects of inserting and deleting transactions. Thirty of the axioms have been shown to be non-contradictory. The remaining 21 are believed to be non-contradictory, but the PVS proof is still under development. The 21 un-validated axioms describe the effects of inserting and deleting transactions.

The main source of complexity in the remaining un-validated axioms is the appearance and disappearance of committed delayed transactions in the states of PCI' . For example, if a new transaction is inserted into a path, then the signif-

icant committed delayed transactions already present in that path may or may not disappear depending on the type and location of the newly inserted transaction. This behavior under deleting and inserting transactions made defining axioms which describe the effects of insertion and deletion difficult. If PCI did not include a notion of “leaving trails of transactions” the axiomatization of the abstraction would have been smaller and simpler (but not as interesting).

Refinement Proof. The refinement proof was done by showing that every application of every concrete rule corresponds to some application of an abstract rule. More specifically, we show that for all states s , the abstraction of every state created by every application of every PCI rule to state s is equal to some state created by the application of some PCI' rule to the abstraction of s . Using the inference rules from our theory of PCI, we divided the refinement proof into ten cases—one for each rule.

The PVS proof was developed by a single experienced PVS user in about one month of effort. The final refinement proof required approximately 1000 proof commands (determined by taking the number of lines in the PVS proof file and dividing by two to account for the pretty-printing of right parentheses). While the proof required a significant amount of effort, the refinement proof can be reused for exhaustively verifying other properties of the PCI protocol, as will be shown in section 4.

3.3 Model Checking

We wrote Mur ϕ models of the PCI' protocol over the four network classes. While we intend for the Mur ϕ and PVS models to have the same meaning, there is no formal relationship between the Mur ϕ models and the PVS model of the PCI' protocol. The Mur ϕ modeling language allows natural expression of a protocol defined using rules. The Mur ϕ models were an average of 670 lines long (including comments). The Mur ϕ code for the PCI' d-commit rule is given below. The rule is part of a larger rule-set in which the subscripts i and j range over all positions in all paths in the network. Mur ϕ rule-sets provide a convenient notation for quantifying rules. In the Mur ϕ model, transactions are written dr, fr, drc and frc to denote “data read,” and “flag read” transactions. A c at the end of a transaction indicates that the transaction has been committed.

```
Rule "d Commit 1"
  (uncommitted (trans)) &
  ((trans = dr) -> (! (member (path, fr) | member (path, frc))))
==>
begin delete_trans (path, commit (trans)); -- delete t_t
  network[i][j] := commit (trans);
end;
```

The precondition checks that the transaction is uncommitted, the additional predicate on dr encodes one of the preconditions to the PC property. The action, given after the “ $==>$ ”, replaces the transaction with its committed form.

We defined two pieces of auxiliary state to encode the PC property as a safety property. The auxiliary state stored the completion order of the read and write

Network	CPU Time	States
A	35.35 sec.	1614
B	18.68 sec.	914
C	12.56 sec.	648
D	51.20 sec.	2690
Total	117.79 sec.	5866

Fig. 5. Model checking results for PC property on PCI' .

transactions at their respective destination agents. Using the completion orders stored in the auxiliary state, we can state the PC property as a safety property which reads:

If the data read transaction has completed at the data address, then either flag write occurred after the flag read, or the flag read has not completed, or the data read occurred after the data write.

This safety property is stated as an invariant of the $Mur\phi$ model of PCI' and checked using explicit state enumeration. The time and number of states required to check each reduced network is given in table 3.3. No violations were found in any of the networks. The $Mur\phi$ models were developed and verified in about one week of effort by a new $Mur\phi$ user (including time to install and learn $Mur\phi$).

4 Verifying PC for PCI_e and an SOC Property

After using the abstraction to verify the PC property for PCI extended with local master IDs, we decided to verify the PCI_e as originally defined without local master IDs. We expected that this experiment would reveal a violation of the PC property due to completion stealing. We also verified the PCI protocol for another transaction ordering property related to packet chains used in the VSI for SOC applications. We now describe each experiment in turn.

4.1 Verifying PC on PCI_e without Local Master IDs

We created a new reduced model, PCI'_e , so that PCI_e is a trace inclusion refinement of PCI'_e . Removing local master IDs in PCI_e meant that all completions look the same and we could no longer distinguish between related and unrelated completions relative to the property being verified in the PCI'_e model. This meant that completions from other agents removed by the abstraction must be allowed to appear and disappear at any location at any time in the PCI'_e model. We did build a PVS proof showing that PCI_e refines PCI'_e since the changes from PCI to PCI_e are minor.

We created a $Mur\phi$ model of PCI'_e with a restriction that committed transactions could only appear and disappear at the head or tail of a path (to limit state explosion), and checked the model using breadth first state traversal. A violation of the PC invariant was found at a depth of 16 states after checking

38,436 states in just under ten minutes. The resulting error trace correlated to a PCI_e error trace involving completion stealing, as reported by Corella [Cor96]. In general, trace inclusion refinement does not require that violations in the PCI'_e model imply violations in the PCI_e model, however, in this case we were lucky. The $\text{Mur}\phi$ model of PCI'_e was created by modifying one of the PCI' models.

4.2 PCI Transaction Ordering for SOC

Next, we verified the property that “only posted transactions are always received in the order they were sent.” This in-order property is useful for packet chains transmitted between two virtual components using the VCI on-chip bus transported over a PCI network. The VCI on-chip bus standard requires that packet chains transmitted between VCI components must be received in the order they were transmitted regardless of what communication medium is used to transport them.

In the case of the in-order property, there are only two agents in the property: the sender and receiver. Since the in-order property uses only two agents, there is only one network symmetry class to consider (rather than four as required for the PC property). We re-used the $\text{Mur}\phi$ model of PCI' used in the PC verification to check all traces of the abstract network and found (as expected) that only posted transactions, or posted transactions followed by a single delayed transaction, are received in order over PCI networks. Order is not guaranteed for any other combination of PCI transactions. Re-using the PC $\text{Mur}\phi$ model to check the in-order property required changing the invariant and the topology of the network. No other changes to the refinement proof or model were needed.

5 Related Work

We compare our work to two other bodies of work: other PCI verification work and other complete verification methods for parameterized or unbounded systems. Our work is done at the same level of abstraction as the liveness work done by Corella [CSZ97]. However, we consider transaction ordering rather than liveness and we use a combination of theorem proving and model checking rather than a manual proof. The PCI specification was used in a limited sense in a case study to demonstrate a BDD abstraction technique by Clarke [CJLW99]. An inconsistency in a state machine in the PCI specification was found by them in a network consisting of a master, an agent and a bus. In our PCI work, we consider all PCI networks and we reason about transaction ordering rather than message response. Finally, the work reported here extends our own previous work [MHG98,MHJG00] in that we have now exhaustively verified all execution traces of all PCI networks. This exhaustive verification depends on an abstraction which was hinted at in [MHJG00], but which we present here in detail and justify using a mechanical refinement proof.

Although many techniques for reasoning about parameterized or unbounded systems exist, we found none as well-suited for reasoning about transaction ordering properties of non-trivial protocols over unbounded branching networks.

Das [DDP99] presents a method which requires the user to supply a set of predicates to split the state space into abstract states which are constructed iteratively using BDDs to represent the reachable states. Abdulla [AAB⁺99] addresses the problem of verifying systems operating on infinite data structures using a combination of automatically generated abstractions and symbolic reachability analysis.

Our method is most similar in aim to that presented by Kesten et al using predicate transformers for reasoning about properties on unbounded branching networks [KMM⁺97] applied to the Futurebus+ protocol. At the bus/bridge level, Futurebus+ and PCI are similar in that they allow arbitrary branching networks of processes. Predicate transformers over regular string expressions were used to reason about single-bus instances of the Futurebus+ protocol. While predicate transformers over regular *tree* expressions can be used to reason about properties over arbitrary branching networks, to the best of our knowledge, such an extension has not yet been reported. We have presented an abstraction technique that handles multiple bus instances of the PCI protocol.

Our technique for reasoning about networks is also similar to that of Bhargavan et al [BOG00] in which a combination of HOL theorem proving and SPIN model checking is used to reason about properties of several routing information protocols including the Bellman-Ford algorithm and a protocol under development for ad hoc mobile networks. Bhargavan's work does not include the requirement of acyclic networks but is specifically targeted at routing information protocols rather than message passing protocols.

6 Concluding Remarks

Based on our experiences in this case study, we offer the following observations about the application of formal reasoning to the verification of non-trivial protocols over branching networks:

- Theorem proving or model checking applied *directly* to the problem is respectively too difficult or too complex. We found that identifying and proving invariants using a direct theorem proving approach required a prohibitive amount of manual effort. Although model checking applied directly to PCI is not possible given the unbounded number of network topologies, as reported in [MHJG00], we found that even modest coverage of the large, but finite, execution space of several concrete networks using a model checker was beyond the capacity of current model checking tools.
- The abstraction based on network symmetry modulo path lengths was essential but required significant modifications to the PCI protocol. These modifications required a refinement proof to show that the modified protocol preserved the safety properties of the original protocol.
- Expressing the operational semantics of PCI and PCI' using rules provided a single convenient notation for both the refinement proof and model checking. We re-emphasize that a total of 10 rules, which can be printing on one page, were required to specify the behavior of the entire PCI protocol. In addition,

translating the PVS theory of the PCI' rules used in the refinement proof into a set of rules for a Mur ϕ model required little effort. Creating a Mur ϕ model of PCI' from the PCI' rules required about one week of effort by a new Mur ϕ user.

- The current abstraction and refinement proof result in a model that can be exhaustively examining very quickly, however, deriving the abstraction and showing refinement required a great deal of manual effort.

Based on the above observations, we are pursuing general methods for reasoning about parameterized systems in acyclic branching topologies based on abstractions that are “correct by construction” in the sense that the resulting abstraction will not require a refinement proof. We anticipate that this abstraction technique will require more CPU time to check the reduced model, but will require less user time to show refinement. At present, we are investigating the use of predicate abstraction and regular expressions to represent the state of each component and nonisomorphic Steiner trees over labeled terminals to cover the network symmetry classes.

References

- AAB⁺99. P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhench. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [HP99].
- AL91. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
- All00. VSI Alliance. Technical documents and specifications, Accessed April 2000. Available online at <http://www.vsi.org/library/specs/summary.html>.
- BOG00. Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Routing information protocol in HOL/SPIN. In *Theorem Provers in Higher-Order Logics 2000: TPHOLs00*, August 2000.
- CJLW99. Edmund Clarke, Somesh Jha, Yuan Lu, and Dong Wang. Abstract BDDs: A technique for using abstraction in model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1999. Springer-Verlag.
- Cor96. Francisco Corella. Proposal to fix ordering problem in PCI 2.1, 1996. Accessed April 2000. <http://www.pcisig.com/reflector/thrd8.html\#00706>.
- CSZ97. Francisco Corella, Robert Shaw, and Cui Zhang. A formal proof of absence of deadlock for any acyclic network of PCI buses. In *Computer Hardware Description Languages, CHDL'97*, Toledo, Spain, April 1997.
- DDP99. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99].
- HP99. Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

- ID96. C. Norris Ip and David L. Dill. Verifying systems with replicated components in Mur ϕ . In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- Jon00. Michael Jones. PCI transaction ordering case study. <http://www.cs.utah.edu/~mjones/pci-pvs.html>, 2000.
- KMM⁺97. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. symbolic model checking with rich assertional languages. In Orna Grumberg, editor, *Computer-Aided Verification, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.
- MHG98. Abdel Mokkedem, Ravi Hosabettu, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, November 1998.
- MHJG00. Abdel Mokkedem, Ravi Hosabettu, Michael D. Jones, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. *Formal Methods in Systems Design*, 16(1):93–119, January 2000.
- PCI95. PCISIG. PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1, June 1995.

Visualizing System Factorizations with Behavior Tables^{*}


Alex Tsow and Steven D. Johnson

System Design Methods Laboratory
Computer Science Department
Indiana University
`atsow@cs.indiana.edu`

Abstract. Behavior tables are a design formalization intended to support interactive design derivation for hardware and embedded systems. It is a reformulation of the DDD transformation system, bridging behavioral and architectural forms of expression. The tabular representations aid in visualizing design aspects that are subject to interactive refinement and optimization. These ideas are illustrated for system factorization, an import class of decompositions used in design derivation. A series of examples shows how features seen in the behavior tables determine the course of a factorization and how the rules of a core behavior table algebra compose into the more large scale transformations done at the interactive level.


1 Introduction

We are developing a formal reasoning system for synchronous system design based on *behavior table* notation and equivalence preserving transformations. This project stems from research embodied in the *DDD transformation system*, an interactive design derivation system based on functional algebra. Behavior tables have been used informally with DDD for a long time. This experience, together with new tools and results in requirements analysis, has motivated us to look at behavior tables as a formal basis for design derivation, that is, as the object of formal manipulation, rather than merely a display notation.

This paper illustrates how behavior tables help in visualizing design transformations. We focus on a particular class of transformations we call *system factorizations*, which are very important in DDD-style derivations. A system factorization is a kind of architectural decomposition, breaking one system description into two or more connected subsystems. The term “factorization” is appropriate because the decomposition involves a distributivity law for conditional expressions . Applications of system factorization range from simple component isolation, analogous to *function allocation* in high level synthesis, to abstract data-type encapsulation, analogous to **architecture** declarations in VHDL.

^{*} This research is supported, in part, by the National Science Foundation under Grant MIP9610358.

In our practice, behavior tables are an intermediate notation. They come into play as the designer is developing an architecture for a behavioral specification. The tables are poor at expressing either algorithmic behavior or architectural organization. Their value lies just in their neutrality between control and structure dominated views of the design.

Interactive visualization is enhanced by the use of color. In the printed version of this paper we use background shading to indicate features that would be colored by a graphical tool. The electronic version  includes colorization.






The behavior table for a JK flip-flop with synchronous preset, right, illustrates the most important syntactic features. The primary interpretation is that of a synchronous sequential process, mapping input streams (infinite sequences) to output streams. In this case, for boolean type \mathcal{B} , $JK: \mathcal{B}^\infty \times \mathcal{B}^\infty \times \mathcal{B}^\infty \rightarrow \mathcal{B}^\infty \times \mathcal{B}^\infty$, as indicated in the topmost bar of the table. The three left-hand columns comprise a *decision table* listing all the cases for inputs P, J, and K; \natural is a *don't care* symbol.

JK: (J,K,P) → (Q, <u>R</u>)					
P	J	K	Q	<u>R</u>	
0	\natural	\natural	1	\overline{Q}	
\natural	0	0	Q	\overline{Q}	
\natural	0	1	0	\overline{Q}	
\natural	1	0	1	\overline{Q}	
\natural	1	1	\overline{Q}	\overline{Q}	

The two right-hand columns comprise an *action table* giving the values for *signals* Q and R. Signal names heading the action table columns are of two kinds. Q is *sequential*, denoting a state-holding abstract register; the terms in Q's column specify its next-state values. R is *combinational*, (indicated by the underline); the terms in R's columns specify its current-state values. That is, $Q_0 = \natural$, $\underline{R}_k = Q_k$ and

$$Q_{k+1} = \begin{cases} 1 & \text{if } P_k = 0; \text{ otherwise} \\ Q_k & \text{if } J_k = K_k = 0 \\ 0 & \text{if } J_k = 0 \text{ and } K_k = 1 \\ 1 & \text{if } J_k = 1 \text{ and } K_k = 0 \\ \overline{Q_k} & \text{if } J_k = K_k = 1 \end{cases}$$

This example also shows that behavior tables are not efficient expressions of architecture. Signal R is invariantly the negation of signal Q, as is seen in the redundant entries in its column of the action table. At the same time, the fact that this pattern is so plainly *seen* is an example of tables' value in design visualization.

After a background review in Sec. , Sec.  gives a brief syntactic and semantic outline of behavior tables and its core algebra. More details can be found in . In Sec. , we demonstrate various factorizations of a “shift-and-add” multiplier, showing how basic visualizations appear in behavior table forms. The section ends with a larger scale example, encapsulating the abstract memory component of a garbage collector. This example shows the scale at which behavior tables have proven effective for design analysis. Section  outlines directions for further development.

2 Background

Behavior tables arose as an informal notation used in the DDD transformation system, a formal system which represents designs as functional modeling expressions [1]. In DDD, control oriented behavioral expressions are simultaneous systems of tail-recursive function definitions, each function being a control point. Architectural expressions are recursive systems of streams definitions, representing a network of subsystem connections and their observable values over time. A central step in a design derivation is a construction translating between these two modes of expression. Most transformations performed prior to this step deal with control aspects, while most performed afterward are either architectural manipulations or data refinements.

A higher level design task will involve both control and architecture. For example, in behavioral synthesis, scheduling involves control, allocation involves architecture, and register binding lies in between. Hence, in a typical design derivation, there may be a considerable distance between a behavioral manipulation such as scheduling and the architectural manipulations it *anticipates* [2].

As our case studies grew, we began printing our specifications in a tabular form in order to plan our derivation strategies. The methodology progressed, and the tables became integral to the design process, although they were still treated simply as a display format. The underlying formal manipulations were still performed on recursive expressions. In the early 1990s we began to consider basing the DDD formal system on tables rather than expressions [3].

The emergence of table-based tools and notations in requirements analysis, such as *Tablewise* [4], *SCR** [5], *RSML's and-or tables* [6], and *PVS table expressions* [7], provided further encouragement. There is a good deal of evidence for tables' value in visualizing complex control problems and as a means for systematic case analysis, especially when the tool is augmented by automated analyses.

We propose a table based tool and sketch behavior table semantics in [8] and describe its core algebra in [9]. As in many formal reasoning tools, the idea is to secure a sound and near-complete kernel of basic identities from which higher level transformations can be composed. The core algebra for behavior tables corresponds to a set of five kernel structural transformations for DDD, first described by Johnson [10].

These very basic rules are too finely grained to be useful in interactive reasoning, or, for that matter, in automated synthesis. They serve instead as an basis for establishing the soundness of higher level rules. Mathematical soundness is established by the fact that higher transformations can be implemented by compositions of the core rules. Although we have not done it yet, the demonstration of this claim will be in the form of a tool built over a core transformation "engine" implementing the more complex higher level transformations.

Of course, nothing precludes the automated use of behavior tables in a *formal synthesis* fashion [11], in which an optimization algorithm drives the derivation. However, we do not know whether tables are a useful representation for that purpose. We are specifically interested in interactive aspects of design derivation

and in how the tables serve to visualize design goals and automated analyses. We would expect this tool to be used at a relatively high level in synchronous-reactive design, both for hardware and embedded software applications.

3 Behavior Table Expressions

Behavior tables are closed expressions composed of first order terms over a user specified algebraic structure (e.g. a many sorted Σ algebra). Variables come from three disjoint sets: inputs I , sequential or *register* signals S , and combinational signals C . Our notion of term evaluation is standard: the value of a term, t , is written $\sigma \llbracket t \rrbracket$, where σ is an *assignment* or association of values to variables. A behavior table has the form:

<i>Name: Inputs \rightarrow Outputs</i>	
<i>Conditions</i>	<i>Registers and Signals</i>
\vdots	\vdots
<i>Guard</i>	<i>Computation Step</i>
\vdots	\vdots

Inputs is a list of input variables and *Outputs* is a subset of the registered and combinational variables. *Conditions* is a set of terms ranging over finite types, such as truth values, token sets, etc. The *guards* are tuples of constants denoting values that the conditions can take. The conditions together with the guards form a *decision table*. Each guard indexes a *computation step* or *action*. An action is tuple of terms, each corresponding to a combinational or registered variable. The actions and the internal signal names (i.e. non input variables) compose the *action table*.

A *behavior table* denotes a relation between infinite input and output sequences, or *streams*. Suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I . Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \dots \rangle$ for *ISC* as follows:

- (a) $\sigma_n(i)$ is given for all $i \in I$ and all n .
- (b) For each $s \in S$, $\sigma_0(s) = x_s$.
- (c) $\sigma_{n+1}(s) = \sigma_n \llbracket t_{s,k} \rrbracket$ if guard g_k holds for σ_n .
- (d) For each $c \in C$, $\sigma_n(c) = \sigma_n \llbracket t_{c,k} \rrbracket$ if guard g_k holds for σ_n .

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \dots \rangle$. This semantic relation is well defined in the absence of circular dependencies amongst combinational actions $\{t_{c,k} \mid c \in C, g_k \in G\}$. The relation is a function (i.e. deterministic) when the branches of the decision table are exhaustive and exclusive. We shall restrict our attention to behavior tables that are *well formed* in these respects. Well formedness reflects the usual quality required of synchronous digital systems, finite state machines, etc.

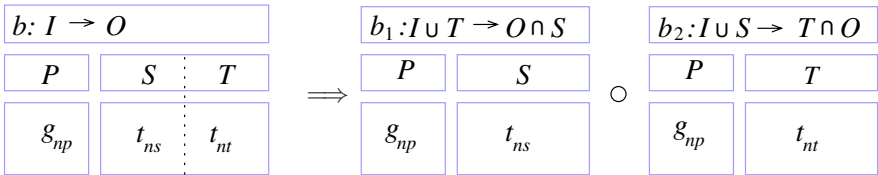
Behavior tables denote persistent, communicating processes, rather than sub-procedures. Consequently behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Composition is specified by giving a *connection map* that is faithful to each component's arity. In our function-oriented modeling notation, such compositions are expressed as named recursive systems of equations,

$$\begin{aligned} S(U_1, \dots, U_n) &= (V_1, \dots, V_m) \text{ where} \\ (X_{11}, \dots, X_{1q_1}) &= \mathcal{T}_1(W_{11}, \dots, W_{1\ell_1}) \\ &\vdots \\ (X_{p1}, \dots, X_{pq_p}) &= \mathcal{T}_p(W_{p1}, \dots, W_{p\ell_p}) \end{aligned}$$

in which the defined variables X_{ij} are all distinct, each \mathcal{T}_k is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

The core behavior table algebra presented in [1] has eleven rules. We shall discuss three of them here, to give a sense of what they are like. The core algebra for structural manipulation in DDD had only five rules [1], one of which was later shown by Miner to be unnecessary. There are more rules for behavior tables for several reasons. Behavior tables have decomposition and hiding rules whose counterparts in DDD are subsumed by general laws functional algebra. Behavior tables have rules that affect decision tables and action tables, while no such distinction exists in DDD's system expressions. Behavior table rules apply to isolated rows as well as columns, while the corresponding DDD rules, in affect, apply only to columns. Finally, the purpose of the rules is to be a basis for automation, not mathematical minimality.

The *Decomposition Rule* is central to system factorization. It splits one table into two, both having the same decision part, by introducing the free variables thus created to the I/O signatures.



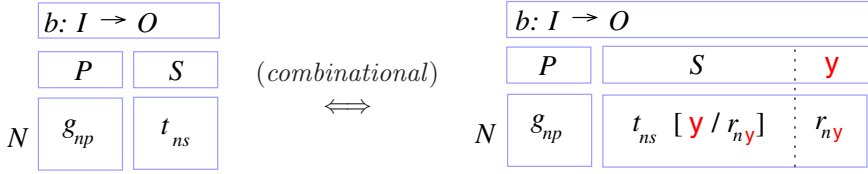
The composition operator, ‘ \circ ’ represents a connection combinator,

$$b(I) \equiv O \text{ where } \begin{aligned} O_1 &= b_1(I_1) \\ O_2 &= b_2(I_2) \end{aligned}$$

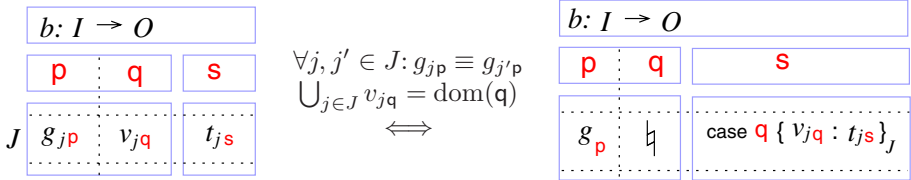
in which $I_1 = I \cup T$, $I_2 = I \cup S$, $O_1 = O \cap S$, and $O_2 = O \cap T$.

The *Identification Rule*, below, introduces and uses a name for a subexpression. In the rule scheme below, guards g_{np} and terms t_{ns} on the left-hand side are indexed by the sets of row numbers N , predicates P , and signals S . On the right, a new signal, y , is introduced. with action terms $\{r_{ny}\}_{n \in N}$. In each cell

of the action table, should y 's defining term occur in t_{ns} , it can be replaced by y . Conversely, a column of an action table can be eliminated, applying the rule from right to left, by substituting y 's defining terms for y throughout the rest of the table.



Our third example is the *Conversion Rule*, whose effect is to coalesce rows in the decision table part. This is done by incorporating a selection expression in the action table that chooses the appropriate signal value. The side conditions say that all of the other guarding values in the decision table must be identical, and each possibility for the condition q must be accounted for.



As these examples illustrate, we have developed a notation of *table schemes* to express the core identities. In these schemes, the boxes play the role of quantifiers. Capitalized variables are index sets, lower case variables range over sets of the same name, and sans serif letters are dummy constants.



S

Thus, the form $R \boxed{x_{rs}}$ represents $\{g_{r,s} \mid r \in R \text{ and } s \in S\}$.

4 System Factorization and Decomposition

System factorization is the organization of systems into architectural components. In the context of behavior tables this corresponds to their decomposition into a connection hierarchy in which behavior tables are the leaves. The motivations for factorization include: isolating functions in need of further specification, encapsulating abstract data types, grouping complementary functionality, targeting known components, and separating system control from system data

A designer has three decisions to make when factoring a given system component: which functions and signals to factor, how to group functionality in the presence scheduling conflicts, and assigning usage of communication lines between components. Behavior tables facilitate the this process by coloring terms relevant to each choice. Some of these decisions are characterized as optimization problems; we can automate more of the process by solving these problems.

The first three examples below decompose a behavior table describing a “shift-and-add” multiplication algorithm. The ASM chart  and timing diagram from Fig.  specifies its algorithm and interface respectively, while DDD automatically constructs the behavior table `MULT`. We introduce function and signal factorization with this example. The last example combines these techniques to isolate an abstract memory from a “stop and copy” garbage collector.

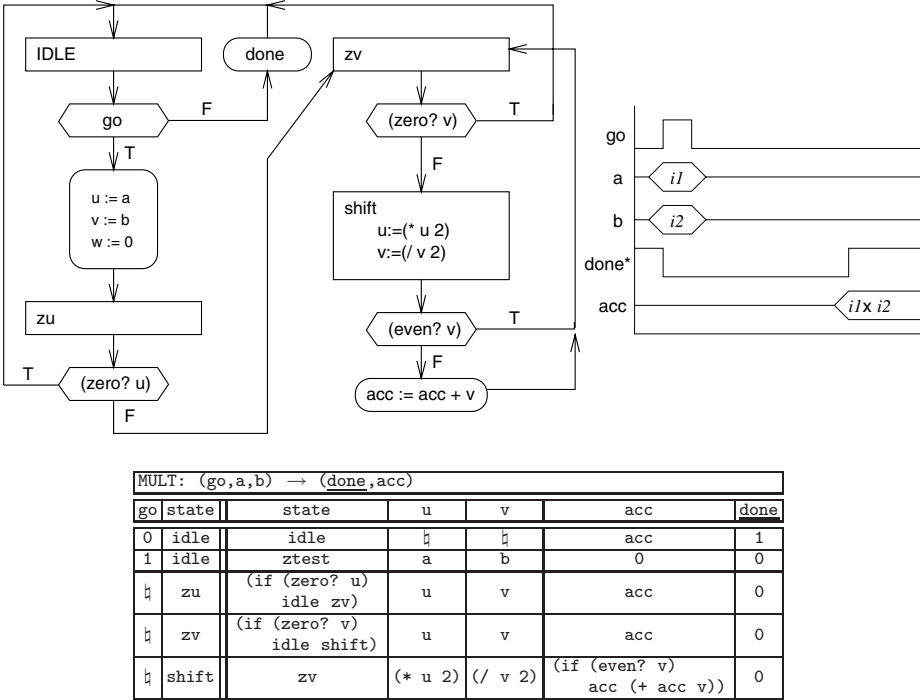



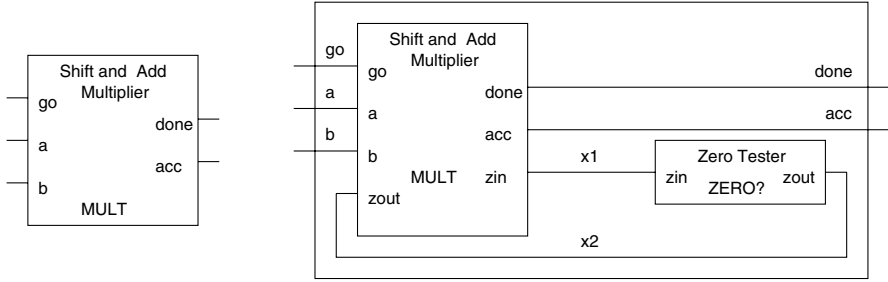
Fig. 1. Algorithmic, protocol, and behavior table specification of a shift-and-add multiplier

4.1 Single Function Factorization

The simplest variety of system factorization separates instances of one particular function f from a sequential system S . The decomposition produces two tables: a trivial table F that unconditionally applies f to its inputs, and a modified system S' where instances of f are replaced with the F ’s output and where new combinational signals have been created to carry the arguments of f to F .

We demonstrate this type of factorization by isolating the function `zero?` from the multiplier in Fig. . The block diagram view summarizes this archi-

tectural refinement. The connections hierarchy is maintained using the lambda expression below:



$$\begin{aligned} &\lambda(go, a, b).(done, acc) \text{ where} \\ &\quad (done, acc, x1) = MULT(go, a, b, x2) \\ &\quad (x2) = ZERO?(x1) \end{aligned}$$

For the remainder of the examples, we assign I/O signals of separate blocks the same name to imply connection.

The *specification set* defines the function(s) to be factored; it is $\{zero?\}$ for this illustration. Guiding selection of *subject terms*, the instances of functions from the specification set the user intends to factor, the table below highlights all applications of **zero?**. We make no refinement to this initial selection of subject terms.

Tabular notation conveys scheduling information because the actions of each row occur in the same semantic step. The color scheme identifies scheduling collisions (i.e. rows with multiple subject term) by using a different color to render the conflicting terms. This example has consistent scheduling because each use of **zero?** occurs in a different row; we display these terms with a black background.

MULT: (go,a,b) → (done,acc)						
go	state	state	u	v	acc	done
0	idle	idle	⊔	⊔	acc	1
1	idle	ztest	a	b	0	0
⊔	zu	(if (zero? u) idle zv)	u	v	acc	0
⊔	zv	(if (zero? v) idle shift)	u	v	acc	0
⊔	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

The transformation augments the original behavior table **MULT** with a new combinational signal **zin** to carry the actual arguments to **zero?**. Signal **zin** becomes an output of **MULT** and is connected to the input channel of **ZERO?**, the table encapsulating the function **zero?**. **ZERO?** is a trivial behavior table because it has no decision table; it simply places the value of (**zero? zin**) on its output **zout**. The decomposition routes **zout** back to **MULT** and replaces the subject terms with the signal **zout**. The changes to **MULT** are highlighted with dark grey in the result of our decomposition below.

MULT: (go,a,b, zout) → (<u>done</u> ,acc, <u>zin</u>)						
go	state	state	u	v	acc	<u>done</u> <u>zin</u>
0	idle	idle	h	h	acc	1 h
1	idle	ztest	a	b	0	0 h
h	zu	(if zout idle zv)	u	v	acc	0 u
h	zv	(if zout idle shift)	u	v	acc	0 v
h	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0 h

<u>ZERO?:(zin)→(zout)</u>
<u>So*</u>
<u>(zero? zin)</u>

4.2 Factoring Multiple Functions

We can generalize factorization to permit the encapsulation of several functions. This transformation follows the single function form, but also introduces a communication line to transmit control. Given a stream system S and a specification set $\{f_1, \dots, f_n\}$, the transformation produces the following:

- a modified system S' containing combinational output signals for each argument of every f_i and a combinational output *inst* to select the operation; the subject terms are replaced with the appropriate input channels
- a table F using the *inst* signal to select the required function from the specification set

The initial form of the transformation makes inefficient use of communication lines by assigning dedicated input and output lines to each f_i . The following example illustrates how to remedy this using behavior tables to illuminate optimization opportunities.

The system in Fig. 1 is the starting point for this transformation, and the block diagram in Fig. 1 shows the connections between the new MULT (block defined by the dotted boundary) and the isolated component, ALU. We now use the expanded specification set $\{\text{zero?}, \text{even?}, \text{add}\}$. To assist with selecting subject terms, the instances of function application from the specification set are highlighted according to their scheduling: a light grey background with a black foreground indicates parallel applications, while a black background with a white foreground indicates a single application.

The table shows that the initial subject terms $(+ \text{acc } v)$ and $(\text{even? } v)$ execute simultaneously. The designer may resolve the conflict by refining the selection of subject terms, but may also allow parallel applications; this decision should be guided by a knowledge of complementary functionality. For didactic reasons, we choose to eliminate this scheduling collision by removing the $(\text{even? } v)$ from the set of subject terms.

MULT: (go,a,b) → (done,acc)						
go	state	state	u	v	acc	done
0	idle	idle	⊔	⊔	acc	1
1	idle	ztest	a	b	0	0
⊔	zu	(if (zero? u) idle zv)	u	v	acc	0
⊔	zv	(if (zero? v) idle shift)	u	v	acc	0
⊔	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

Following the decomposition above, combinational output lines `i1`, `i2` and `i3` are added to carry arguments to `zero?` and `+`. Additionally, the table acquires a combinational output `inst` used to select the functionality of the encapsulated component. The new inputs to `MULT`, `out1` and `out2` replace subject terms corresponding to `zero?` and `+` respectively. The new table, `ALU` uses the `inst` input to determine which operation to perform; each function has its own input and output signals. As before modifications to the `MULT` table have a white foreground over a dark grey background.

MULT: (go,a,b, out1 , out2) → (done,acc, i1 , i2 , i3 , inst)											
go	state	state	u	v	acc	done	i1	i2	i3	inst	
0	idle	idle	⊔	⊔	acc	1	⊔	⊔	⊔	⊔	
1	idle	ztest	a	b	0	0	⊔	⊔	⊔	⊔	
⊔	zu	(if out1 idle zv)	u	v	acc	0	u	⊔	⊔	zero	
⊔	zv	(if out1 idle shift)	u	v	acc	0	v	⊔	⊔	zero	
⊔	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	⊔	acc	v	add	

ALU: (i1,i2,i3,inst)→(out1,out2)		
inst	out1	out2
zero	(zero? i1)	⊔
add	⊔	(+ i2 i3)

This initial decomposition is very conservative about the use of input and output lines. An automated minimization algorithm could be run here, but we also want to provide the user with an interface to manually control this process; a designer may want to preserve various inputs and outputs for specific uses such as “address” or “data”.

The tables aid the user with the collection of compatible input and output signals. Highlighting the nontrivial terms (values besides `⊔`) of `i1`, `i2`, and `i3`, we see that the present scheduling allows the values on `i1` and `i2` to be sent on a single output line.

MULT: (go,a,b,out1,out2) \rightarrow (done,acc, i1 , i2 , i3 ,inst)										
go	state	state	u	v	acc	done	i1	i2	i3	inst
0	idle	idle	<i>h</i>	<i>h</i>	acc	1	<i>h</i>	<i>h</i>	<i>h</i>	<i>h</i>
1	idle	ztest	a	b	0	0	<i>h</i>	<i>h</i>	<i>h</i>	<i>h</i>
<i>h</i>	zu	(if out1 idle zv)	u	v	acc	0	u	<i>h</i>	<i>h</i>	zero
<i>h</i>	zv	(if out1 idle shift)	u	v	acc	0	v	<i>h</i>	<i>h</i>	zero
<i>h</i>	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	<i>h</i>	acc	v	add

MULT: (go,a,b,out1,out2) \rightarrow (done,acc, i1 , i2 ,inst)										
go	state	state	u	v	acc	done	i1	i2	inst	
0	idle	idle	<i>h</i>	<i>h</i>	acc	1	<i>h</i>	<i>h</i>	<i>h</i>	
1	idle	ztest	a	b	0	0	<i>h</i>	<i>h</i>	<i>h</i>	
<i>h</i>	zu	(if out1 idle zv)	u	v	acc	0	u	<i>h</i>	zero	
<i>h</i>	zv	(if out1 idle shift)	u	v	acc	0	v	<i>h</i>	zero	
<i>h</i>	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	acc	v	add	

Similarly, coloring the nontrivial terms on out1 and out2 suggests that the the results of (zero? i1) and (+ i1 i2) can be returned on the same output channel. However, these terms have different types preventing this optimization. Later in the design pipeline, reducing integers to boolean vectors enables this optimization.

ALU: (i1,i2,inst) \rightarrow (out1 , out2)				
inst	out1		out2	
zero	(zero? i1)	<i>h</i>		
add	<i>h</i>	(add i1 i2)		

4.3 Signal Factorization

The next family of factorizations encapsulates state. Given a stream system S and a specification set of *signals* $\{v_1, \dots, v_n\}$, the transformation removes the columns v_i from S and introduces a control signal *inst* to create S' . To create the isolated component ST , signal factorization constructs a table using the signals v_i as the entries in its action table and *inst* in the decision table to enumerate the possible updates to v_i .

Beginning this example where the one from Sec. 4.2 ends, signals **u** and **v** determine the specification set. Intuitively, factoring these signals yields a pair of shift registers. Figure 4.1 shows this decomposition within the dotted box. Since the subject terms in a signal factorization are always the columns of the specified signals, we reserve the limited palette for identifying updates to **u** and **v**. Light grey, black, and dark grey correspond to assignment of new variables **a** and **b** to **u** and **v**, the preservation of **u** and **v**, and the binary left and right shift of **u** and **v**. Subject terms are italicized.

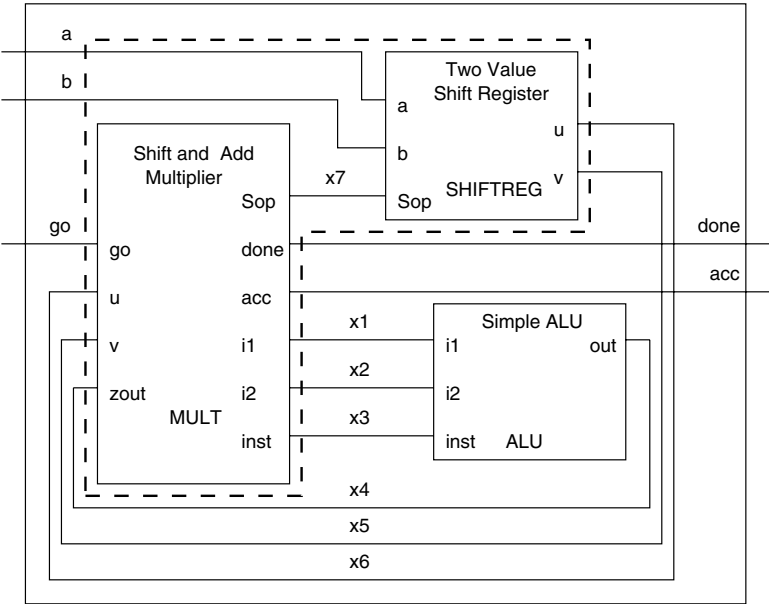


Fig. 2. Result of successive factorizations in Sec.  and Sec. 

MULT: (go,a,b,out) → (done,acc,i1,i2,inst)									
go	state	state	u	v	acc	done	i1	i2	inst
0	idle	idle	⊔	⊔	acc	1	⊔	⊔	⊔
1	idle	ztest	<i>a</i>	<i>b</i>	0	0	⊔	⊔	⊔
⊔	zu	(if out1 idle zv)	<i>u</i>	<i>v</i>	acc	0	<i>u</i>	⊔	zero
⊔	zv	(if out1 idle shift)	<i>u</i>	<i>v</i>	acc	0	<i>v</i>	⊔	zero
⊔	shift	zv	(<i>* u 2</i>)	(<i>/ v 2</i>)	(if (even? v) acc out2)	0	acc	<i>v</i>	add

The new signal in MULT, *Sop*, carries the instruction tokens *init*, *hold*, and *shift* corresponding to the signal updates above. The isolated component SHIFTREG acquires its action table from the columns *u* and *v* in MULT. Apart from *Sop*, the only inputs to SHIFTREG are *a* and *b* –the unresolved variables appearing in the terms of its action table. Since MULT no longer requires *a* and *b*, they are eliminated from its input signature. Changes to MULT are italicized.

MULT: (go,out,u,v) → (done,acc,i1,i2,inst,Sop)									
go	state	state	<i>Sop</i>	acc	done	i1	i2	inst	
0	idle	idle	⊔	acc	1	⊔	⊔	⊔	SHIFTREG: (a,b,Sop) → (u,v)
1	idle	ztest	<i>init</i>	0	0	⊔	⊔	⊔	
⊔	zu	(if out1 idle zv)	<i>hold</i>	acc	0	<i>u</i>	⊔	zero	
⊔	zv	(if out1 idle shift)	<i>hold</i>	acc	0	<i>v</i>	⊔	zero	
⊔	shift	zv	<i>shift</i>	(if (even? v) acc out2)	0	acc	<i>v</i>	add	

<i>Sop</i>	<i>u</i>	<i>v</i>
<i>init</i>	<i>a</i>	<i>b</i>
<i>hold</i>	<i>u</i>	<i>v</i>
<i>shift</i>	(<i>* u 2</i>)	(<i>/ v 2</i>)

4.4 The Stop and Copy Garbage Collector

Figure 10.1 is the behavior table for a garbage collector. Its “stop and copy” algorithm uses two memory half-spaces represented by abstract registers OLD and NEW. The goal of this factorization is to hide the memory data abstraction in a separate component.

OLD and NEW, together with their methods, **rd** and **wt**, define the specification set. The corresponding subject terms are highlighted in Fig. 10.1 with either black or light grey. As a combination of signal and function factorization, the instruction tokens select both state update and return value. References to OLD and NEW cannot appear in the resulting version of GC; consequently instruction naming subsumes their role as parameters to **rd** and **wt**. To illustrate this, we have shaded the instances of the “write NEW; read OLD” instruction (**Mwnro**) with light grey.

Like the previous examples, the initial decomposition supplies dedicated input and output signals for each operation of the factored component. Figure 10.2 displays the results, and indicates altered or new terms with dark and light grey. The dark grey shows the replacement of terms calling **rd** and also guides the optimization of new signals, while the light grey shows instances of the **Mwnro**.

MEM: (Mwoi1, Mwoi2, Mwni1, Mwni2, Mroi, Mrni, Mop) → (MR0out, MRNout)				
Mop	OLD	NEW	MR0out	MRNout
Mnop	OLD	NEW	⊔	⊔
Mswap	NEW	OLD	⊔	⊔
Mwold	(wt OLD Mwoi1 Mwoi2)	NEW	⊔	⊔
Mwnew	OLD	(wt NEW Mwni1 Mwni2)	⊔	⊔
Mroid	OLD	NEW	(rd OLD Mroi)	⊔
Mrnew	OLD	NEW	⊔	(rd NEW Mrni)
Mwnro	OLD	(wt NEW Mwni1 Mwni2)	(rd OLD Mroi)	⊔

We complete the factorization by collecting compatibly typed inputs and outputs when possible. Figure 10.3 shows these final input signals, and the table below displays the condensed output signals.

MEM: (MwtAd, MData, MRdAd, Mop) → (Mout)			
Mop	OLD	NEW	Mout
Mnop	OLD	NEW	⊔
Mswap	NEW	OLD	⊔
Mwold	(wt OLD MwtAd MData)	NEW	⊔
Mwnew	OLD	(wt NEW MwtAd MData)	⊔
Mroid	OLD	NEW	(rd OLD MRdAd)
Mrnew	OLD	NEW	(rd NEW MRdAd)
Mwnro	OLD	(wt NEW MwtAd MData)	(rd OLD MRdAd)

A realization of this design used a dual-ported DRAM capable of performing the **Mwnro** instruction in just a little over one DRAM cycle [1].

[illegible]

Fig. 3. Behavior Table for a Garbage Collector

GC: (Rd) → (AK)																
MON	Rd											AK	MvAd	MvTa	MvId	MvOp
		(= U A)	(pointer? H)	(bvec-h? H)	(eq? forward (tag d))	(tag H)	(= C 0)	(= C 1)	MON		H	D	C	U	A	
1	idle	x	x	x	x	x	x	x	driver	H	D	C		Mout	0	H
2	"	0	x	x	x	x	x	x	idle	H					1	Mold
3	driver	x	x	x	x	x	x	x	idle	0	D	C		U	1	MvOp
4	"	x	0	x	x	x	x	x	nextobj		D	C		U	0	MvOp
5	nextobj	x	x	x	x	x	x	x	objtype	H	Mout	C		U	0	MvOp
6	"	x	x	0	1	x	x	x	driver	H	Mout	C			0	Mold
7	"	x	x	0	0	x	x	x	driver	H	Mout	C			0	Mold
8	objtype	x	x	x	x	x	x	x	driver	H	D	C			0	MvOp
9	"	x	x	x	x	0	fixed	x	copy	(cell H (addi-ptr (pr-pt H)))	D	(addi-2 (fixed-size H))	U	0	H	Mold
10	"	x	x	x	0	vec	x	x	vec	(addi-ptr (pr-pt H))	D	(btow-c (pr-pt d))	U	0	H	Mold
11	"	x	x	x	0	bvec	x	x	vec	(cell H (addi-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	0	H	Mold
12	vec	x	x	x	x	x	1	x	driver	(cell H (addi-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	0	A	MvOp
13	"	x	x	x	x	x	x	x	copy	(addi-ptr (pr-pt H))	Mout	C	U	0	A	MvOp
14	copy	x	x	x	x	x	x	x	driver	H	D	C			0	MvOp
15	"	x	x	x	x	x	x	x	copy	(cell H (addi-ptr (pr-pt H)))	Mout	(sub1 c)	U	0	A	MvOp

Fig. 5. End Result of Signal Factorization

5 Further Work

Truth tables, decision tables, register transfer tables, and many similar forms are commonplace in design practice. Their already evident utility suggests that tabular notations have a role in design visualization and should, therefore, be considered for interfacing with formal reasoning tools. In our experience with interactive digital design derivation, tables have already proven helpful in proof planning and design analysis, but our previous automation has involved a shallow embedding of tabular syntax in an expression oriented modeling notation.

We are working toward a graphical interface with highlighting and animation facilities to support interactive visualization and proof management. This paper has illustrated ways in which these facilities are used in decomposition tasks. We demonstrated several kinds of factorizations, ranging from simple function allocation to abstract data type encapsulation. However, this is by no means the full range of decompositions needed.

The core algebra underlying the factorization transformations of this paper is not adequate to implement all the constructions of a high level synthesis system. They are not able to perform scheduling, for example, or even retiming. The DDD system uses fold/unfold transformations on control oriented behavior expressions to perform scheduling refinements, but these remain to be incorporated in the behavior table laws.

The reason is that we are still trying to determine suitable “levels” of equivalence. All the laws of [1] are bisimulation preserving, too strong a relationship for many kinds of design optimizations, such as scheduling. On the other hand, weaker forms of bisimulation may still be too strong to account for transformations that incorporate transaction protocols [2].

Heuristics are another topic of continuing study. Our research emphasizes interaction in order to make design processes observable, but it does not promote manual design over automation. Among the main uses of system factorization are targeting reusable subsystems, IP cores, and off-the-shelf components. These goal directed activities propagate constraints back into the main design. We are interested in exploring how visualization techniques, such as those of Sec. 4 help designers solve such problems.

References

1. FMCAD 2000. <http://link.springer.de/series/lncs/>.
2. Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994. Technical Report No. 456, 155 pages, <http://ftp.cs.indiana.edu/pub/techreports/TR456.ps.gz>.
3. Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.
4. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

5. D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, National Aeronautics and Space Administration Langley Research Center (NASA/LRC), Hampton VA 23681-0001, November 1994.
6. Steven D. Johnson. Manipulating logical organization with system factorizations. In Leiser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.
7. Steven D. Johnson. A tabular language for system design. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Lfm97: Fourth NASA Langley Formal Methods Workshop*, September 1997. NASA Conference Publication 3356, <http://atb-www.larc.nasa.gov/Lfm97/>.
8. Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).
9. Steven D. Johnson and Alex Tsow. Algebra of behavior tables. In C. M. Holloway, editor, *Lfm2000*. Langley Formal Methods Group, NASA, 2000. Proceedings of the 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia, 13-15 June, 2000, <http://shemesh.larc.nasa.gov/lm/Lfm2000/>.
10. Ramayya Kumar, Christian Blumenröhr, Dirk Eisenbiegler, and Detlef Schmid. Formal synthesis in circuit design – a classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design*, pages 294–309, Berlin, 1996. Springer LNCS 1166. Proceedings of FMCAD’96.
11. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
12. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Revised May 1996. Available, with specification files, from URL <http://www.csl.sri.com/csl-95-12.htm>.
13. Franklin P. Prosser and David E. Winkel. *The Art of Digital Design*. Prentice/Hall International, second edition, 1987.
14. Kamlesh Rath. *Sequential System Decomposition*. PhD thesis, Computer Science Department, Indiana University, USA, 1995. Technical Report No. 457, 90 pages.
15. Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 736–740. IEEE, November 1993.
16. M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI95)*, pages 86–89. IEEE, March 1995.

Author Index

- Aagaard, Mark D., 126, 263
Alur, Rajeev, 55
Amla, Nina, 283
Ashar, Pranav, 354
- Basin, David, 91
Ben-David, Shoham, 390
Bjesse, Per, 372
Bloem, Roderick, 37, 143
Bonifácio, Adilson Luiz, 316
Bryant, Randal E., 470, 486
- Cerny, Eduard, 299
Cerone, Antonio, 246
Chauhan, Pankaj, 486
Claessen, Koen, 372
Clarke, Edmund M., 197, 486
Cook, Byron, 126
- Day, Nancy A., 126
Dean, Mark E., 1
Dill, David L., 335, 470
- Emerson, E. Allen, 283
- Friedrich, Stefan, 91
- Gabow, Harold N., 37
German, S., 197
Goel, Amit, 486
Gopalakrishnan, Ganesh, 505
Grosu, Radu, 55
Grumberg, Orna, 390
Gupta, Aarti, 354
Gupta, Anubhav, 354
- Hachtel, Gary D., 73
Hamaguchi, Kiyoharu, 455
Harrison, John, 217
Heyman, Tamir, 390
Hou, Jin, 299
Hu, Alan J., 335
Hunt, Warren A., Jr., 234
- Johnson, Steven D., 520
Jones, Michael, 505
Jones, Robert B., 263
- Kashiwabara, Toshinobu, 455
Kurshan, Robert P., 283
- Lu, Y., 197
- Manolios, Panagiotis, 161
Meinel, Christoph, 443
Melham, Thomas F., 263
Milne, George J., 246
Mödersheim, Sebastian, 91
Moon, In-Ho, 73
- Namjoshi, Kedar S., 283
- O’Leary, John W., 263
- Pace, Gordon J., 405
- Ravi, Kavita, 143
Reif, Wolfgang, 179
Ritter, Gerd, 423
Ruf, Jürgen, 179
Russinoff, David M., 3
- Sawada, Jun, 234
Schellhorn, Gerhard, 179
Schuster, Assaf, 390
Seger, Carl-Johan H., 263
Sheeran, Mary, 108
Shimizu, Kanna, 335
Singh, Satnam, 108
Somenzi, Fabio, 37, 73, 143
Stålmarmark, Gunnar, 108
Stangier, Christian, 443
- Tsow, Alex, 520
- Urushihara, Hidekazu, 455
- Veith, Helmuth, 197
Vieira Moura, Arnaldo, 316
Vollmer, Tobias, 179
- Wang, Bow-Yaw, 55
Wang, D., 197
Wilson, Chris, 470
- Yang, Zijiang, 354